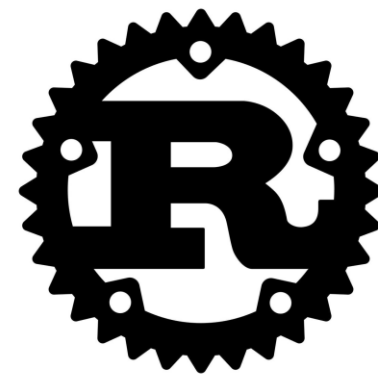


建立多執行緒網頁伺服器

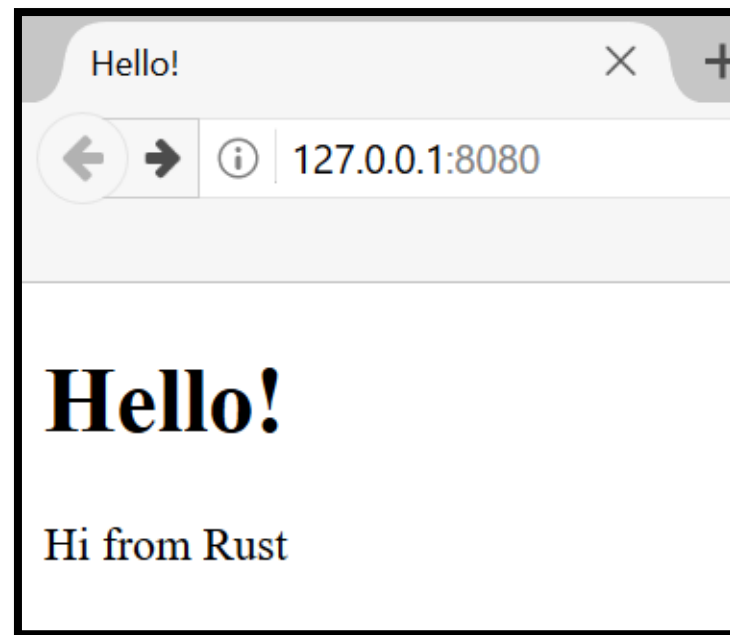


建立多執行緒網頁伺服器

在專案中將建立一個可以回覆「hello」的網頁伺服器，如右邊圖示：

以下是建構網頁伺服器的計劃：

1. 學習一些 TCP 與 HTTP。
2. 在socket上監聽 TCP 連線。
3. 解析一些的 HTTP 請求。
4. 建立合適的回應。
5. 透過執行緒池(thread pool)改善伺服器的負載。



不過在開始之前，需要提醒一件事，使用的方法不會是在 Rust 中建立網頁伺服器的最佳方案。

crates.io 上有不少已經能用在正式環境的 crate，它們都有提供比所建立的還更完善的網頁伺服器與執行緒池。

因為 Rust 是個系統程式設計語言，可以選擇想運用的抽象層級，而且可以比其他語言更可能且實際地抵達最底層。會寫出基本的 HTTP 伺服器與執行緒池，來幫助瞭解往後可能會用到的 crate 背後的基本概念與技術。

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 會先從建立一個可運作的單一執行緒網頁伺服器作為起始。在開始之前，先快速瞭解一下建構網頁伺服器會涉及到的協定。
- 網頁伺服器會涉及到兩大協定為超文本傳輸協定(Hypertext Transfer Protocol, HTTP)與傳輸控制協定(Transmission Control Protocol, TCP)。這兩種協定都是請求-回應(request-response)協定，這代表客戶端會初始一個請求，然後伺服器接聽請求並提供回應給客戶端。協定會定義這些請求與回應的內容。
- TCP 是個較底層的協定並描述資訊如何從一個伺服器傳送到另一個伺服器的細節，但是它並不指定資訊內容為何。HTTP 建立在 TCP 之上並定義請求與回應的內容。
- 技術上來說，HTTP 是可以與其他協定組合的，但是對大多數場合中，HTTP 主要還是透過 TCP 來傳送資訊。會處理 TCP 與 HTTP 中請求與回應的原始位元組(raw bytes)。

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 監聽 TCP 連線

- 網頁伺服器需要監聽一個 TCP 連線，所以這就是要處理的第一個步驟。標準函式庫有提供 `std::net` 模組能使用。如往常一樣建立一個新的專案：

```
$ cargo new hello
    Created binary (application) `hello` project
$ cd hello
```

- 輸入底下範例的程式碼到 `src/main.rs`。此程式碼會監聽 `127.0.0.1:7878` 本地位址(address)傳來的TCP流(Stream)。當其收到傳入的資料流時，它就會印出**連線建立！**。

```
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        println!("連線建立!");
    }
}
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 監聽 TCP 連線

- 透過 TcpListener，可以監聽 127.0.0.1:7878 位址上的 TCP 連線。在位址中，分號之前指的是代表電腦自己的 IP 位址，然後 7878 是通訊埠(port)。選擇此通訊埠的原因有兩個：HTTP 通常不會佔用此通訊埠，所以伺服器不太可能會與機器上執行的其他網路伺服器衝突。
- 在此情境中的 bind 函式與常見的 new 函式行為類似，這會回傳一個新的 TcpListener 實例。此函式會叫做bind 的原因是因為在網際網路中，連接一個通訊埠並監聽就稱為「綁定(bind)通訊埠」。
- bind 函式會回傳 Result<T, E>，也就是說綁定可能會失敗。舉例來說，連接通訊埠 80 需要管理員權限(非管理員使用者可以連接 1023 以上的通訊埠)，所以如果不是管理員卻嘗試連接通訊埠 80 的話，綁定就不會成功。
- 另一個例子是，如果執行同個程式碼兩次產生兩個實例，造成這兩個程式同時監聽同個通訊埠的話，綁定也不會成功。由於只會寫個用於學習目的的基本伺服器，不太需要擔心如何處理這些種類的錯誤。所以如果遇到錯誤的話，採用 unwrap 來停止程式。

```
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        println!("連線建立!");
    }
}
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 監聽 TCP 連線

- TcpListener 的 incoming 方法會回傳一個疊代器，給予一連串的流(更準確的來說是 TcpStream 型別的流)。一個流(Stream)代表的是客戶端與伺服器之間的開啟的連線。
- 而連線(connection)指的是整個請求與回應的過程，這之中客戶端會連線至伺服器、伺服器會產生回應，然後伺服器會關閉連線。這樣一來，就能讀取 TcpStream 來查看客戶端傳送了什麼，然後將回應寫入流中，將資料傳回給客戶端。整體來說，此 for 迴圈會依序遍歷每個連線，然後產生一系列的流使其能夠加以處理。
- 目前處理流的方式包含呼叫 unwrap，這當流有任何錯誤時，就會結束程式。如果沒有任何錯誤的話，程式就會顯示訊息。會在下個範例在成功的情況下加入更多功能。
- 當客戶端連接伺服器時，可能會從 incoming 方法取得錯誤的原因是因為實際上不是在遍歷每個連線。反之，是在走訪連線嘗試。連線不成功可能有很多原因，而其中許多都與作業系統有關。舉例來說，許多作業系統都會限制它們能支援的同時連線開啟次數，當新的連線超出此範圍時就會產生錯誤，直到有些連線被關閉為止。

```
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        println!("連線建立!");
    }
}
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 監聽 TCP 連線

```
Running `target/debug/hello`  
連線建立！  
連線建立！  
連線建立！
```

- 跑跑看此程式碼吧！在終端機呼叫 `cargo run` 然後在網頁瀏覽器載入 `127.0.0.1:7878`。瀏覽器應該會顯示一個像是「Connection reset」之類的錯誤訊息，因為伺服器目前還不會回傳任何資料。但是當觀察終端機，在瀏覽器連接伺服器時，應該會看到一些訊息顯示出來！
- 有時候可能從一次瀏覽器請求看到數個訊息顯示出來，原因很可能是因為瀏覽器除了請求頁面內容以外，也嘗試請求其他資源，像是出現在瀏覽器分頁上的 `favicon.ico` 圖示。
- 而瀏覽器嘗試多次連線至伺服器的原因還有可能是因為伺服器沒有回傳任何資料。當 `stream` 離開作用域並在迴圈結尾被釋放時，連線會在 `drop` 的實作中被關閉。瀏覽器有時處理被關閉的連線的方式是在重試幾次，因為發生的問題可能是暫時的。不管如何，現在最重要的是成功取得 TCP 的連線了！
- 當執行完特定版本的程式碼後，記得按下 `ctrl-c` 來停止程式。然後在變更一些程式碼後重新呼叫 `cargo run` 來確保有執行到最新的程式碼。

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 讀取請求
 - 來實作讀取瀏覽器請求的功能吧！為了能分開取得連線的方式與處理連線的方式，會建立另一個新的函式來處理連線。
 - 在此 `handle_connection` 新的函式中，會讀取從 TCP 流取得的資料並顯示出來，使其可以觀察瀏覽器傳送的資料。請修改程式碼成右邊範例：

```
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    println!("請求: {:#?}", http_request);
}
```


建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 讀取請求
 - 將 `std::io::prelude` 和 `std::io::BufReader` 引入作用域來取得特定的特徵，使其可以讀取並寫入流之中。
 - 在 `main` 函式的 `for` 迴圈中，不同於印出取得連線的訊息，現在會呼叫新的 `handle_connection` 函式並將 `stream` 傳入。
 - 在 `handle_connection` 函式中，建立 `BufReader` 的實例並取得 `stream` 的可變參考。
 - `BufReader` 提供了緩衝區(buffering)，幫助管理 `std::io::Read` 方法的呼叫。

```
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    println!("請求: {:#?}", http_request);
}
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 讀取請求
 - 建立了一個變數 `http_request` 來收集瀏覽器傳送到伺服器的每行請求。加上 `Vec<_>` 型別詮釋來指示想要將每行收集成向量。
 - `BufReader` 實作的 `std::io::BufRead` 特徵有提供個 `lines` 方法。該方法會回傳個 `Result<String, std::io::Error>` 的疊代器，這會在每次看到換行 (`newline`) 位元組時，將資料流分開。
 - 用 `map` 對每個 `Result` 呼叫 `unwrap` 來取得 `String`。如果資料不是有效的 UTF-8 或是讀取流時發生問題的話，`Result` 可能會產生錯誤。

```
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    println!("請求: {:#?}", http_request);
}
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 讀取請求
 - 在正式環境的程式應該要適當地處理這些錯誤，但為了簡潔在這裡選擇直接在遇到錯誤時就終止程式。
 - 瀏覽器會傳送兩次換行字元來表達 HTTP 的請求結束了，所以要確定從流中取得一個請求的話，**就重複取得行數直到有一行是空字串為止**。
 - 一旦將行數收集到向量中，就使用好看的除錯格式印出來，使其可以觀察瀏覽器傳送了哪些指令給伺服器。

```
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
};

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    for stream in listener.incoming() {
        let stream = stream.unwrap();

        handle_connection(stream);
    }
}

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    println!("請求: {:#?}", http_request);
}
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 讀取請求
 - 讓嘗試看看此程式碼！開啟程式並再次從網頁瀏覽器發送請求。注意到仍然會在瀏覽器中取得錯誤頁面，但是程式在終端機的輸出應該會類似以下結果：

依據瀏覽器，可能會看到一些不同的輸出結果。現在顯示了請求的資料，可以觀察為何瀏覽器會產生多次請求，可以看看 **Request: GET** 之後的路徑。如果重複的連線都在請求/的話，就能知道瀏覽器在重複嘗試取得/，因為它沒有從程式取得回應。拆開此請求資料來理解瀏覽器在向程式請求什麼。

```
$ cargo run
Compiling hello v0.1.0 (file:///projects/hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.42s
Running `target/debug/hello`
Request: [
  "GET / HTTP/1.1",
  "Host: 127.0.0.1:7878",
  "User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:99.0) Gecko/20100101 Firefox/99.0",
  "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8",
  "Accept-Language: en-US,en;q=0.5",
  "Accept-Encoding: gzip, deflate, br",
  "DNT: 1",
  "Connection: keep-alive",
  "Upgrade-Insecure-Requests: 1",
  "Sec-Fetch-Dest: document",
  "Sec-Fetch-Mode: navigate",
  "Sec-Fetch-Site: none",
  "Sec-Fetch-User: ?1",
  "Cache-Control: max-age=0",
]
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 仔細觀察 HTTP 請求

- HTTP 是基於文字的協定，而請求格式如下：

```
Method Request-URI HTTP-Version CRLF
headers CRLF
message-body
```

- 第一行是請求行(request line)並持有客戶端想請求什麼的資訊。請求行的第一個部分代表著想使用的方法(method)，像是 GET 或 POST，這描述了客戶端如何產生此請求。在此例中，客戶端使用的是 GET 請求。
 - 請求行的下一個部分是 /，這代表客戶端請求的統一資源標誌符(Uniform Resource Identifier，URI)，URI 絕大多數(但不是絕對)就等於統一資源定位符(Uniform Resource Locator，URL)。
 - URI 與 URL 的差別並不重要，但是 HTTP 規格使用的術語是 URI，所以這裡將 URL 替換為 URI。

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

```
Method Request-URI HTTP-Version CRLF
headers CRLF
message-body
```

- 仔細觀察 HTTP 請求
 - 最後一個部分是客戶端使用的 HTTP 版本，然後請求行最後以 CRLF 序列做結尾，CRLF 指的是輸入(carriage return)與換行(line feed)，這是打字機時代的術語！CRLF 序列也可以寫成 \r\n，\r 指的是輸入，而 \n 指的是換行。CRLF 序列將請求行與剩餘的請求資料區隔開來。注意到當 CRLF 印出時，會看到的是新的一行而不是 \r\n。
 - 觀察目前從程式中取得的請求行資料，看到它使用 GET 方法、/ 為請求 URI 然後版本為 HTTP/1.1。
 - 在請求行之後，剩餘從 Host: 開始的行數都是標頭(header)。GET 請求不會有本體(body)。
 - 可以嘗試看看從不同的瀏覽器或訪問不同的位址，像是 127.0.0.1:7878/test，來看看請求資料有什麼改變。
 - 現在知道瀏覽器在請求什麼了，來回傳一些資料吧！

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 寫入回應

- 現在要實作傳送資料來回應客戶端的請求。回應格式如下：

```
HTTP-Version Status-Code Reason-Phrase CRLF
headers CRLF
message-body
```

- 第一行為狀態行(status line)，這包含回應使用的 HTTP 版本、用來總結請求結果的狀態碼，以及狀態碼的文字來描述原因。在 CRLF 序列後，會接著任何標頭、另一個 CRLF 序列，然後是回應的本體。
 - 以下是個使用 HTTP 版本 1.1 的回應範例，其狀態碼為 200、文字描述為 OK，沒有標頭與本體：

```
HTTP/1.1 200 OK\r\n\r\n
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 寫入回應

```
HTTP/1.1 200 OK\r\n\r\n
```

```
fn handle_connection(mut stream: TcpStream) {  
    let buf_reader = BufReader::new(&mut stream);  
    let http_request: Vec<_> = buf_reader  
        .lines()  
        .map(|result| result.unwrap())  
        .take_while(|line| !line.is_empty())  
        .collect();  
  
    let response = "HTTP/1.1 200 OK\r\n\r\n";  
  
    stream.write_all(response.as_bytes()).unwrap();  
}
```

- 狀態碼 200 是標準的成功回應。這段文字就是小小的 HTTP 成功回應。將此寫入流中作為對成功請求的回應吧！在 `handle_connection` 函式中，移除原先印出請求資料的 `println!`，然後換成右上範例的程式碼：
- 新的第一行定義了變數 `response` 會持有成功訊息的資料。然後對 `response` 呼叫 `as_bytes` 來將字串轉換成位元組。`stream` 中的 `write_all` 方法接收 `&[u8]` 然後將這些位元組直接傳到連線中。由於 `write_all` 操作可能會失敗，如前面一樣對任何錯誤使用 `unwrap`。同樣地，在實際的應用程式中，應該要在此加上錯誤處理。
- 有了這些改變，執行程式碼然後下達請求。不再顯示任何資料到終端機上了，所以不會看到任何輸出，只會有 Cargo 執行的訊息。當網頁瀏覽器讀取 `127.0.0.1:7878` 時，應該會得到一個空白頁面，而不是錯誤了。剛剛手寫了一個 HTTP 請求與回應！

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 回傳真正的 HTML
 - 實作不止是回傳空白頁面的功能。
 - 首先在專案根目錄建立一個檔案 `hello.html`，而不是在 `src` 目錄內。可以輸入任何想要的 HTML，如右邊範例：
 - 這是最小化的 HTML 文件，其附有一個標頭與一些文字。為了要在收到請求後從伺服器回傳此檔案，要修改之前範例的 `handle_connection` 來讀取 HTML 檔案、加進回應本體中然後傳送出去：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Hi from Rust</p>
  </body>
</html>
```

```
use std::fs;
// --省略--

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    let status_line = "HTTP/1.1 200 OK";
    let contents = fs::read_to_string("hello.html").unwrap();
    let length = contents.len();

    let response =
        format!("{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}");

    stream.write_all(response.as_bytes()).unwrap();
}
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 回傳真正的 HTML
 - 在 use 新增了 fs 來將標準函式庫中的檔案系統模組引入作用域。
 - 接下來，使用 format! 來加入檔案內容來作為成功回應的本體。
 - 為了確保這是有效的 HTTP 回應，加上 Content-Length 標頭並設置為回應本體的大小，在此例中就是 hello.html 的大小。

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Hi from Rust</p>
  </body>
</html>
```

```
use std::fs;
// --省略--

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    let status_line = "HTTP/1.1 200 OK";
    let contents = fs::read_to_string("hello.html").unwrap();
    let length = contents.len();

    let response =
        format!("{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}");

    stream.write_all(response.as_bytes()).unwrap();
}
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 回傳真正的 HTML
 - 透過 `cargo run` 執行此程式碼並在瀏覽器讀取 `127.0.0.1:7878`，應該就會看到 HTML 的顯示結果了！
 - 目前忽略了 `http_request` 中的請求資料，並毫無條件地回傳 HTML 檔案內容。
 - 這意味著如果嘗試在瀏覽器中請求 `127.0.0.1:7878/something-else`，還是會得到相同的 HTML 回應。這樣伺服器是很受限的，而且這也不是大多數網頁伺服器會做的行為。想要依據請求自訂回應，並只對格式良好的 / 請求回傳 HTML 檔案。

```
use std::fs;
// --省略--

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let http_request: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();

    let status_line = "HTTP/1.1 200 OK";
    let contents = fs::read_to_string("hello.html").unwrap();
    let length = contents.len();

    let response =
        format!("{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}");

    stream.write_all(response.as_bytes()).unwrap();
}
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 驗證請求並選擇性地回應
 - 目前網頁伺服器不管客戶端的請求為何，都會回傳 HTML 檔案。加個功能來在回傳 HTML 檔案前檢查瀏覽器請求是否為 /，如果瀏覽器請求的是其他的話就回傳錯誤。為此得修改 `handle_connection` 成像是右上範例這樣。此新的程式碼會檢查收到的請求，比較是否符合 / 的前半部分，並增加了 `if` 與 `else` 區塊來處理不同請求：
 - 只會查看 HTTP 請求的第一行，所以與其讀取整個請求到向量中，不如呼叫 `next` 來取得疊代器的第一個項目就好。第一個 `unwrap` 會處理 `Option`，如果疊代器沒有任何項目的話程式就會停止。第二個 `unwrap` 處理的則是 `Result`，它和之前範例 `map` 裡的 `unwrap` 有相同的效果。
 - 接著，檢查 `request_line` 是否等同於以 / 路徑形式寫成的 `GET` 請求。如果是的話，那麼 `if` 區塊就回傳 HTML 檔案的內容。

```
// --省略--

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let request_line = buf_reader.lines().next().unwrap().unwrap();

    if request_line == "GET / HTTP/1.1" {
        let status_line = "HTTP/1.1 200 OK";
        let contents = fs::read_to_string("hello.html").unwrap();
        let length = contents.len();

        let response = format!(
            "{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}"
        );

        stream.write_all(response.as_bytes()).unwrap();
    } else {
        // some other request
    }
}
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 驗證請求並選擇性地回應
 - 如果 `request_line` 並沒有符合在 `/` 路徑形式下的 `GET` 請求的話，代表收到的是其他請求。稍後會在 `else` 區塊加上回應其他所有請求的程式碼。
 - 執行此程式碼並請求 `127.0.0.1:7878` 的話，應該會收到 `hello.html` 的 `HTML`。如果下達其他任何請求，像是 `127.0.0.1:7878/something-else` 的話，會和執行之前範例的程式碼時獲得一樣的錯誤。

```
// --省略--

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let request_line = buf_reader.lines().next().unwrap().unwrap();

    if request_line == "GET / HTTP/1.1" {
        let status_line = "HTTP/1.1 200 OK";
        let contents = fs::read_to_string("hello.html").unwrap();
        let length = contents.len();

        let response = format!(
            "{}\r\nContent-Length: {}\r\n\r\n{}",
            status_line, length, contents
        );

        stream.write_all(response.as_bytes()).unwrap();
    } else {
        // some other request
    }
}
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 驗證請求並選擇性地回應
- 現在將底下範例的程式碼加入 `else` 區塊來回傳狀態碼為 404 的回應，這代表請求的內容無法找到。也會回傳一個 HTML 頁面讓瀏覽器能顯示並作為終端使用者的回應。

```
// --省略--

fn handle_connection(mut stream: TcpStream) {
    let buf_reader = BufReader::new(&mut stream);
    let request_line = buf_reader.lines().next().unwrap().unwrap();

    if request_line == "GET / HTTP/1.1" {
        let status_line = "HTTP/1.1 200 OK";
        let contents = fs::read_to_string("hello.html").unwrap();
        let length = contents.len();

        let response = format!(
            "{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}"
        );

        stream.write_all(response.as_bytes()).unwrap();
    } else {
        // some other request
    }
}
```

```
// --省略--
} else {
    let status_line = "HTTP/1.1 404 NOT FOUND";
    let contents = fs::read_to_string("404.html").unwrap();
    let length = contents.len();

    let response = format!(
        "{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}"
    );

    stream.write_all(response.as_bytes()).unwrap();
}
```

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 驗證請求並選擇性地回應
 - 在此的狀態行有狀態碼 404 與原因描述 NOT FOUND。回應的本體會是 404.html 檔案內的 HTML。會需要在 hello.html 旁建立一個 404.html 檔案來作為錯誤頁面。
 - 同樣地，可以使用任何想使用的 HTML 或者使用底下範例的 HTML 範本：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello!</title>
  </head>
  <body>
    <h1>Oops!</h1>
    <p>Sorry, I don't know what you're asking for.</p>
  </body>
</html>
```

```
// --省略--
} else {
  let status_line = "HTTP/1.1 404 NOT FOUND";
  let contents = fs::read_to_string("404.html").unwrap();
  let length = contents.len();

  let response = format!(
    "{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}"
  );

  stream.write_all(response.as_bytes()).unwrap();
}
```

有了這些改變後，再次執行的伺服器。請求 127.0.0.1:7878 的話就應該會回傳 hello.html 的內容，而任何其他請求，像是 127.0.0.1:7878/foo 就應該回傳 404.html 的錯誤頁面。

建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 再做一些重構

- 目前 if 與 else 區塊有很多重複的地方，它們都會讀取檔案並將檔案內容寫入流中。唯一不同的地方在於狀態行與檔案名稱。將程式碼變得更簡潔，將不同之處分配給 if 與 else，它們會分別將相對應的狀態行與檔案名稱賦值給變數。就能使用這些變數無條件地讀取檔案並寫入回應。
- 右上範例顯示了替換大段 if 與 else 區塊後的程式碼。
- 現在 if 與 else 區塊只回傳狀態行與檔案名稱的數值至一個元組，可以在 let 陳述式使用模式來解構並將分別兩個數值賦值給 status_line 與 filename，如之前所提及的。

```
// --省略--

fn handle_connection(mut stream: TcpStream) {
    // --省略--

    let (status_line, filename) = if request_line == "GET / HTTP/1.1" {
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();
    let length = contents.len();

    let response =
        format!("{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}");

    stream.write_all(response.as_bytes()).unwrap();
}
```


建立多執行緒網頁伺服器

建立單一執行緒的網頁伺服器

- 再做一些重構

- 之前重複的程式碼現在位於 if 與 else 區塊之外並使用變數 status_line 與 filename。這更容易觀察兩種條件不同的地方，且也意味著如果想要變更讀取檔案與寫入回應的行為的話，只需更新其中一段程式碼就好。與之前範例的程式碼行為一模一樣。
- 現在有個用約莫 40 行 Rust 程式碼寫出的簡單網頁瀏覽器，可以對一種請求回應內容頁面，然後對其他所有請求回應 404 錯誤。
- 伺服器只跑在單一執行緒，這意味著它一次只能處理一個請求。來模擬些緩慢的請求來探討這為何會成為問題。然後會加以修正讓伺服器可以同時處理數個請求。

```
// --省略--

fn handle_connection(mut stream: TcpStream) {
    // --省略--

    let (status_line, filename) = if request_line == "GET / HTTP/1.1" {
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();
    let length = contents.len();

    let response =
        format!("{status_line}\r\nContent-Length: {length}\r\n\r\n{contents}");

    stream.write_all(response.as_bytes()).unwrap();
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 現在的伺服器會依序處理請求，代表它處理完第一個連線之前，都無法處理第二個連線。
- 伺服器收到越來越多請求，這樣的連續處理方式會變得越來越沒效率。
- 如果伺服器收到一個會花很久時間才能處理完成的請求，之後的請求都得等待這個長時間的請求完成才行，就算新的請求能很快處理完成也是如此。
- 需要修正此問題，但首先觀察此問題怎麼發生的。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 對目前伺服器實作模擬緩慢的請求
 - 來觀察看看處理緩慢的請求如何影響目前伺服器實作中的其他請求。
 - 右邊範例實作了處理 `/sleep` 的請求，其在回應前讓伺服器沉睡 5 秒鐘來模擬緩慢的回應。
 - 由於現在有三種情況了，將從 `if` 改成 `match`。需要用字串字面值數值來配對 `request_line`。 `match` 不會像相等方法那樣自動參考和解參考。

```
use std::{
    fs,
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
    thread,
    time::Duration,
};
// --省略--

fn handle_connection(mut stream: TcpStream) {
    // --省略--

    let (status_line, filename) = match &request_line[..] {
        "GET / HTTP/1.1" => ("HTTP/1.1 200 OK", "hello.html"),
        "GET /sleep HTTP/1.1" => {
            thread::sleep(Duration::from_secs(5));
            ("HTTP/1.1 200 OK", "hello.html")
        }
        _ => ("HTTP/1.1 404 NOT FOUND", "404.html"),
    };

    // --省略--
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 對目前伺服器實作模擬緩慢的請求
 - 第一個分支和之前範例的 `if` 區塊相同。第二個分支配對的請求是 `/sleep`。當收到請求時，伺服器會在成功顯示 HTML 頁面之前沈睡 5 秒。第三個和之前範例的 `else` 區塊相同。
 - 可以看出伺服器有多基本：真實的函式庫會以較不冗長的方式來識別處理數種請求！

```
use std::{
    fs,
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
    thread,
    time::Duration,
};
// --省略--

fn handle_connection(mut stream: TcpStream) {
    // --省略--

    let (status_line, filename) = match &request_line[..] {
        "GET / HTTP/1.1" => ("HTTP/1.1 200 OK", "hello.html"),
        "GET /sleep HTTP/1.1" => {
            thread::sleep(Duration::from_secs(5));
            ("HTTP/1.1 200 OK", "hello.html")
        }
        _ => ("HTTP/1.1 404 NOT FOUND", "404.html"),
    };

    // --省略--
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 對目前伺服器實作模擬緩慢的請求
 - 使用 `cargo run` 來啟動伺服器，然後開啟兩個瀏覽器視窗：一個請求 `http://127.0.0.1:7878/` 然後另一個請求 `http://127.0.0.1:7878/sleep`。
 - 如果輸入好幾次 / URI 的話，會如之前一樣迅速地收到回應。但如果先輸入 `/sleep` 在讀取 `/` 的話，會看到 / 得等待 `sleep` 沉睡整整 5 秒鐘後才能讀取。
 - 有好幾種方式能避免緩慢請求造成的請求堆積。其中一種就是要實作的執行緒池(thread pool)。

```
use std::{
    fs,
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream},
    thread,
    time::Duration,
};
// --省略--

fn handle_connection(mut stream: TcpStream) {
    // --省略--

    let (status_line, filename) = match &request_line[..] {
        "GET / HTTP/1.1" => ("HTTP/1.1 200 OK", "hello.html"),
        "GET /sleep HTTP/1.1" => {
            thread::sleep(Duration::from_secs(5));
            ("HTTP/1.1 200 OK", "hello.html")
        }
        _ => ("HTTP/1.1 404 NOT FOUND", "404.html"),
    };

    // --省略--
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 對目前伺服器實作模擬緩慢的請求
- 透過執行緒池改善負載
 - 執行緒池(thread pool)會產生一群執行緒來等待並隨時準備好處理任務。
 - 當程式收到新任務時，它會將此任務分配給執行緒池其中一條執行緒，然後該執行緒就會處理該任務。池中剩餘的執行緒在第一條執行緒處理任務時，仍能隨時處理任何其他來臨的任務。
 - 當第一條執行緒處理完成時，會回到閒置執行緒池之中，等待處理新的任務。執行緒池能並行處理連線，增加伺服器的負載。
 - 會限制執行緒池的數量為少量的數量就好，以避免造成阻斷服務(Denial of Service，DOS)攻擊。如果程式每次遇到新的請求時就產生新的執行緒，某個人就可以產生一千萬個請求至伺服器，來破壞並用光伺服器的資源，並導致所有請求的處理都被擱置。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過執行緒池改善負載
 - 所以與其產生無限制的執行緒，會有個固定數量的執行緒在池中等待。當有請求來臨時，它們會被送至池中處理。此池會維護一個接收請求的佇列(queue)。
 - 每個執行緒會從此佇列彈出一個請求、處理該請求然後再繼續向佇列索取下一個請求。有了此設計，就可以同時處理 N 個請求，其中 N 就是執行緒的數量。如果每個執行緒都負責到需要長時間處理的請求，隨後的請求還是會阻塞佇列，但是少增加了能夠同時處理長時間請求的數量。
 - 此技巧只是其中一種改善網頁伺服器負載的方式而已。其他可能會探索到的選項還有 fork/join 模型、單執行緒非同步模型或多執行緒非同步模型。如果對此議題有興趣，可以閱讀其他解決方案，並嘗試實作到 Rust 中。像 Rust 這種低階語言，這些所有選項都是可能的。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過執行緒池改善負載
 - 在開始實作執行緒池之前，先討論一下使用該池會是什麼樣子。當嘗試設計程式碼時，先寫出使用者的介面能協助引導設計。寫出程式碼的 API，使其能以所期望的方式呼叫，然後在該結構內實作功能，而不是先實作功能再設計公開 API。
 - 類似於之前所用到的測試驅動開發(test-driven development)，會在此使用編譯器驅動開發方式。會先寫出呼叫所預期函式的程式碼，然後觀察編譯器的錯誤來決定接下來該改變什麼，才能讓程式碼成功運行。不過在那之前，先觀察一些最後不會使用的方式作為起始點。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 對每個請求都產生執行緒
 - 首先探討如果程式碼都對每次連線建立新的執行緒會怎樣。如之前提及的，這不會是最終的計劃，因為這有可能會產生無限條執行緒的問題，但對於討論多執行緒伺服器來說，這是個很好的起始點。
 - 接下來會加入執行緒池來改善，然後比較兩者誰比較簡單。底下範例在 `main` 的 `for` 迴圈中，對每個流都產生一條新的執行緒：

```
fn main() {  
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();  
  
    for stream in listener.incoming() {  
        let stream = stream.unwrap();  
  
        thread::spawn(|| {  
            handle_connection(stream);  
        });  
    }  
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 對每個請求都產生執行緒
 - 如在之前所學到的，`thread::spawn` 會建立一條執行緒並在新的執行緒執行閉包的程式碼。如果執行此程式碼，並在瀏覽器中讀取 `/sleep`，然後在開兩個瀏覽器分頁來讀取 `/` 的話，的確就能看到 `/` 請求不必等待 `/sleep` 完成。但如之前所提的，這最終可能會拖累系統，因為可以無限制地產生新的執行緒。

```
fn main() {  
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();  
  
    for stream in listener.incoming() {  
        let stream = stream.unwrap();  
  
        thread::spawn(|| {  
            handle_connection(stream);  
        });  
    }  
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 建立數量有限的執行緒
 - 想要執行緒池能以類似的方式運作，這樣從執行緒切換成執行緒池時，使用 `API` 的程式碼就不必作出大量修改。底下範例顯示一個想使用的假想 `ThreadPool` 結構體，而非使用 `thread::spawn`：

使用 `ThreadPool::new` 來建立新的執行緒池且有個可設置的執行緒數量參數，在此例中設為4。

然後在 `for` 迴圈中，`pool.execute` 的介面類似於 `thread::spawn`，其會接收一個執行緒池執行在每個流中的閉包。

需要實作 `pool.execute`，使其能接收閉包並傳給池中的執行緒來執行。

此程式碼還不能編譯，但是接下來能試著讓編譯器引導如何修正。

```
fn main() {  
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();  
    let pool = ThreadPool::new(4);  
  
    for stream in listener.incoming() {  
        let stream = stream.unwrap();  
  
        pool.execute(|| {  
            handle_connection(stream);  
        });  
    }  
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過編譯器驅動開發建立 ThreadPool
 - 將之前範例的變更寫入 **src/main.rs**，然後從 cargo check 產生的編譯器錯誤來引導開發吧。

- 以下是第一個收到的錯誤：

```
$ cargo check
   Checking hello v0.1.0 (file:///projects/hello)
error[E0433]: failed to resolve: use of undeclared type `ThreadPool`
--> src/main.rs:11:16
   |
11 |     let pool = ThreadPool::new(4);
   |                   ^^^^^^^^^^^^^ use of undeclared type `ThreadPool`
```

- 很好！此錯誤告訴需要一個 ThreadPool 型別或模組，所以現在就來建立一個。
- ThreadPool 實作會與網頁伺服器相互獨立，所以將 hello crate 從執行檔 crate 轉換成函式庫 crate 來存放 ThreadPool 實作。這樣在切換成函式庫 crate 之後，就能夠將分出來的執行緒池函式庫用在其他想使用執行緒池的地方，而不僅僅是作為網頁請求所用。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過編譯器驅動開發建立 ThreadPool
 - 建立一個包含以下內容的 **src/lib.rs**，這是現在所能寫出最簡單的 ThreadPool 結構體定義了：

```
pub struct ThreadPool;
```

- 然後編輯main.rs檔案將ThreadPool從函式庫crate引入作用域，請將以下程式碼寫入**src/main.rs** 最上方：

```
use hello::ThreadPool;
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過編譯器驅動開發建立 ThreadPool
 - 此程式碼仍然無法執行，再次檢查並取得下一個要解決的錯誤：

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no function or associated item named `new` found for struct `ThreadPool` in the current scope
--> src/main.rs:12:28
12 |         let pool = ThreadPool::new(4);
    |                             ^^^ function or associated item not found in `ThreadPool`
```

- 此錯誤指示需要對 ThreadPool 建立個關聯函式叫做 new。還知道 new 需要有個參數來接受作為引數的 4，並需要回傳 ThreadPool 實例。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過編譯器驅動開發建立 ThreadPool
 - 來實作擁有這些特性的最簡單 new 函式：
 - 選擇 `usize` 作為參數 `size` 的型別，因為知道負數對執行緒數量來說沒有任何意義。也知道 4 會作為執行緒集合的元素個數，這正是使用 `usize` 型別的原因，如同之前段落所講的，再檢查程式碼一次：

```
pub struct ThreadPool;

impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        ThreadPool
    }
}
```

```
$ cargo check
    Checking hello v0.1.0 (file:///projects/hello)
error[E0599]: no method named `execute` found for type `ThreadPool` in the current scope
--> src/main.rs:17:14
17 |         pool.execute(|| {
    |         ^^^^^^^ method not found in `ThreadPool`
```

- 現在錯誤的原因是因為 `ThreadPool` 沒有 `execute` 方法。回想一下「建立數量有限的執行緒」段落中，決定執行緒池要有類似於 `thread::spawn` 的介面。
- 除此之外，會實作 `execute` 函式使其接收給予的閉包並傳至執行緒池中閒置的執行緒來執行。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過編譯器驅動開發建立 ThreadPool

- 定義 ThreadPool 的 execute 方法接收一個閉包來作為參數。回憶一下之前的「Fn 特徵以及將獲取的數值移出閉包」段落中，可以透過三種不同的特徵來接受閉包：Fn、FnMut 與 FnOnce。
- 需要決定這裡該使用何種閉包。知道行為會類似於標準函式庫中 `thread::spawn` 的實作，所以看看 `thread::spawn` 簽名中的參數有哪些界限吧。技術文件會顯示右上結果：
- F 型別參數正是所在意的，T 型別則是與回傳型別有關，而目前並不在意。
- 可以看到 `spawn` 使用 `FnOnce` 作為 F 的界限。這大概就是想要的，因為最終會將 `execute` 的引數傳遞給 `spawn`。現在更確信 `FnOnce` 就是想使用的特徵，**因為執行請求的執行緒只會執行該請求閉包一次**，這正符合 `FnOnce` 中 `Once` 的意思。

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```


建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過編譯器驅動開發建立 ThreadPool

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

- F 型別參數還有個特徵界限 Send 與生命週期界限 'static，這場合中也很實用，需要 Send 來將閉包從一個執行緒轉移到另一個，而會需要 'static 是因為不知道執行緒會處理多久。

- 對 ThreadPool 建立 execute 方法，並採用泛型參數型別 F 與其界限：

```
impl ThreadPool {
    // --省略--
    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
    }
}
```

- 在 FnOnce 之後仍然使用 ()，因為此 FnOnce 代表閉包沒有任何參數且回傳值為單元型別 ()。
- 與函式定義一樣，回傳型別可以在簽名中省略，但是儘管沒有任何參數，還是得加上括號。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過編譯器驅動開發建立 ThreadPool
 - 同樣地，這是 `execute` 方法最簡單的實作，它不會做任何事情，但是指示要先讓程式碼能夠編譯通過。

再次檢查：

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.24s
```

- 編譯通過了！但值得注意的是如果嘗試 `cargo run` 並在瀏覽器下請求的話，會像本章開頭一樣在瀏覽器看到錯誤。函式庫還沒有實際呼叫傳至 `execute` 的閉包！
- 注意：可能聽過對於像是 Haskell 和 Rust 這種嚴格編譯器的語言，會號稱「**如果程式碼能編譯，它就能正確執行。**」但這全然是正確的。專案能編譯，但是它沒有做任何事！如果在寫的是實際的完整專案，這是個寫**單元測試**的好時機，這能檢查程式碼能編譯而且有預期行為。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 在 `new` 驗證執行緒數量
 - 對 `new` 與 `execute` 的參數沒有做任何事情。對這些函式本體實作出所預期的行為吧。
 - 先從 `new` 開始。稍早選擇非帶號型別作為 `size` 的參數，因為負數對於執行緒數量並沒有任何意義。然而，零條執行緒的池一樣也沒有任何意義，但零卻可以是完全合理的 `usize`。
 - 要在回傳 `ThreadPool` 前，加上程式碼來檢查 `size` 有大於零，並透過 `assert!` 來判定。如果為零的話就會恐慌，如右上範例所示：
 - 透過技術文件註解來對 `ThreadPool` 加上技術文件說明。注意到加上一個段落說明何種情況呼叫函式會恐慌，這樣就有遵守良好的技術文件典範，如同之前所討論過的。
 - 嘗試執行 `cargo doc --open` 然後點擊 `ThreadPool` 結構體來看看 `new` 產生出的技術文件長什麼樣子！

```
impl ThreadPool {  
    /// Create a new ThreadPool.  
    ///  
    /// The size is the number of threads in the pool.  
    ///  
    /// # Panics  
    ///  
    /// The `new` function will panic if the size is zero.  
    pub fn new(size: usize) -> ThreadPool {  
        assert!(size > 0);  
  
        ThreadPool  
    }  
  
    // --省略--  
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 在 `new` 驗證執行緒數量
- 除了像這樣使用 `assert!` 巨集之外，也可以將 `new` 改成 `build` 來回傳 `Result`。但是決定在此情況中，嘗試建立零條執行緒的池應該要是不可回復的錯誤。可以試著寫出有以下簽名的 `build` 版本，並比較與 `new` 函式之間的區別：

```
pub fn build(size: usize) -> Result<ThreadPool, PoolCreationError> {
```

```
impl ThreadPool {  
    /// Create a new ThreadPool.  
    ///  
    /// The size is the number of threads in the pool.  
    ///  
    /// # Panics  
    ///  
    /// The `new` function will panic if the size is zero.  
    pub fn new(size: usize) -> ThreadPool {  
        assert!(size > 0);  
  
        ThreadPool  
    }  
  
    // --省略--  
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 建立執行緒的儲存空間
 - 現在有一個有效的執行緒數量能儲存至池中，可以在回傳實例前，建立這些執行緒並儲存至 `ThreadPool` 結構體中。但要怎麼「儲存」執行緒呢？再看一次 `thread::spawn` 的簽名：

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

- `spawn` 函式會回傳 `JoinHandle<T>`，而 `T` 為閉包回傳的型別。試著使用 `JoinHandle` 來看看會發生什麼事。在情況中，傳遞至執行緒池的閉包會處理連線但不會回傳任何值，所以 `T` 就會是單元型別 `()`。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 建立執行緒的儲存空間
 - 右邊範例的程式碼可以編譯，但還不會產生任何執行緒。變更了ThreadPool的定義來儲存一個有 `thread::JoinHandle<()>` 實例的向量，用 `size` 來初始化向量的容量，設置一個會執行些程式碼來建立執行緒的 `for` 迴圈，然後回傳包含它們的 `ThreadPool` 實例。

```
use std::thread;

pub struct ThreadPool {
    threads: Vec<thread::JoinHandle<()>>,
}

impl ThreadPool {
    // --省略--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut threads = Vec::with_capacity(size);

        for _ in 0..size {
            // 將產生些執行緒並儲存至向量
        }

        ThreadPool { threads }
    }

    // --省略--
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 建立執行緒的儲存空間
 - 將 `std::thread` 引入函式庫 `crate` 中的作用域，因為使用 `thread::JoinHandle` 作為 `ThreadPool` 中向量的項目型別。
 - 一旦有收到有效大小，`ThreadPool` 就會建立一個可以儲存 `size` 個項目的新向量。 `with_capacity` 函式會與 `Vec::new` 做同樣的事，但是有一個關鍵差別：它會預先配置空間給向量。
 - 由於知道要儲存 `size` 個元素至向量中，這樣的配置方式會比 `Vec::new` 還要略為有效一點，因為後者只會在元素插入時才重新配置自身大小。
 - 當再次執行 `cargo check`，這次就能成功編譯。

```
use std::thread;

pub struct ThreadPool {
    threads: Vec<thread::JoinHandle<>>,
}

impl ThreadPool {
    // --省略--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut threads = Vec::with_capacity(size);

        for _ in 0..size {
            // 將產生些執行緒並儲存至向量
        }

        ThreadPool { threads }
    }

    // --省略--
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 結構體 `Worker` 負責從 `ThreadPool` 傳遞程式碼給一條執行緒
 - 在之前範例的 `for` 迴圈中留下一個關於建立執行緒的註解。在此該如何實際建立執行緒。
 - 標準函式庫提供 `thread::spawn` 作為建立執行緒的方式，然後 `thread::spawn` 預期在執行緒建立時就會獲得一些程式碼讓執行緒能夠執行。但在這樣的場合中，希望建立執行緒，並讓它們等待之後會傳送的程式碼。標準函式庫的執行緒實作並不包含這種方式，得自己實作。
 - 實作此行為的方法是在 `ThreadPool` 與執行緒間建立一個新的資料結構，這用來管理此新的行為。將此資料結構稱為 `Worker`，這在池實作中是很常見的術語。
 - `Worker` 拿取要執行的程式碼然後在自己的執行緒跑這段程式碼。想像一下這是有一群人在餐廳廚房內工作：工作者(worker)會等待顧客的訂單，然後負責接受這些訂單並完成它們。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 結構體 `Worker` 負責從 `ThreadPool` 傳遞程式碼給一條執行緒
 - 所以與其在執行緒池中儲存 `JoinHandle<()>` 實例的向量，可以儲存 `Worker` 結構體的實例。每個 `Worker` 會儲存一個 `JoinHandle<()>` 實例。然後對 `Worker` 實作一個方法來取得閉包要執行的程式碼，並傳入已經在執行的執行緒來處理。也會給每個 `Worker` 一個 `id`，好在記錄日誌或除錯時，分辨池中不同的工作者。
 - 當建立 `ThreadPool` 時會發生以下事情。會用以下方式在設置完 `Worker` 後，實作將閉包傳遞給執行緒的程式碼：
 1. 定義 `Worker` 結構體存有 `id` 與 `JoinHandle<()>`。
 2. 變更 `ThreadPool` 改儲存 `Worker` 實例的向量。
 3. 定義 `Worker::new` 函式來接收 `id` 數字並回傳一個 `Worker` 實例，其包含該 `id` 與一條具有空閉包的執行緒。
 4. 在 `ThreadPool::new` 中，使用 `for` 迴圈計數來產生 `id`，以此建立對應 `id` 的新 `Worker`，並將其儲存至向量中。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 結構體 `Worker` 負責從 `ThreadPool` 傳遞程式碼給一條執行緒

- 右邊是範例作出修改的方式：

將 `ThreadPool` 中欄位的名稱從 `threads` 改為 `workers`，因為它現在儲存的是 `Worker` 實例而非 `JoinHandle<()>` 實例。

使用 `for` 迴圈的計數作為 `Worker::new` 的引數，然後將每個新的 `Worker` 儲存到名為 `workers` 的向量中。

外部的程式碼(像是在 `src/main.rs` 的伺服器)不需要知道 `ThreadPool` 內部實作細節已經改為使用 `Worker` 結構體，所以讓 `Worker` 結構體與其 `new` 函式維持私有。

`Worker::new` 函式會使用給予的 `id` 並儲存一個 `JoinHandle<()>` 實例，這是用空閉包產生的新執行緒所建立的。

```
use std::thread;

pub struct ThreadPool {
    workers: Vec<Worker>,
}

impl ThreadPool {
    // --省略--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool { workers }
    }
    // --省略--
}

struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize) -> Worker {
        let thread = thread::spawn(|| {});

        Worker { id, thread }
    }
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 結構體 `Worker` 負責從 `ThreadPool` 傳遞程式碼給一條執行緒

- 右邊是範例作出修改的方式：

注意：如果作業系統因為系統資源不足，而無法建立執行緒的話，`thread::spawn` 會恐慌。這會使伺服器恐慌，就算有些執行緒能成功建立。基於簡潔原則，這段程式碼還算能接受。但如果是正式環境的執行緒實作，可能會想使用 `std::thread::Builder` 與其 `spawn` 方法來回傳 `Result`。

此程式碼會編譯通過並透過 `ThreadPool::new` 的指定引數儲存一定數量的 `Worker` 實例。但仍然沒有處理 `execute` 中取得的閉包。看看接下來怎麼做。

```
use std::thread;

pub struct ThreadPool {
    workers: Vec<Worker>,
}

impl ThreadPool {
    // --省略--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool { workers }
    }
    // --省略--
}

struct Worker {
    id: usize,
    thread: thread::JoinHandle<()>,
}

impl Worker {
    fn new(id: usize) -> Worker {
        let thread = thread::spawn(|| {});

        Worker { id, thread }
    }
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過通道傳遞請求給執行緒
 - 接下來要來處理的問題是 `thread::spawn` 中的閉包不會做任何事情。目前透過 `execute` 取得想執行的閉包。但是當在 `ThreadPool` 的產生中建立每個 `Worker` 時，會需要給 `thread::spawn` 一個閉包來執行。
 - 想要建立的 `Worker` 結構體能夠從 `ThreadPool` 中的佇列提取程式碼來執行，並將該程式碼傳至自身的執行緒來執行。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過通道傳遞請求給執行緒
 - 在之前學過的通道(channels)是個能在兩個執行緒間溝通的好辦法，這對專案來說可說是絕佳解法。會用通道來作為任務佇列，然後 `execute` 來傳送從 `ThreadPool` 一份任務至 `Worker` 實例，其就會傳遞該任務給自身的執行緒。以下是計劃：
 1. `ThreadPool` 會建立通道並儲存發送者。
 2. 每個 `Worker` 會持有接收者。
 3. 會建立一個新的結構體 `Job` 來儲存想傳入通道的閉包。
 4. `execute` 方法將會傳送其想執行的 `Job` 至發送者。
 5. 在其執行緒中，`Worker` 會持續走訪接收者並執行它所收到的任何任務閉包。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過通道傳遞請求給執行緒
 - 先在 `ThreadPool::new` 建立通道並讓 `ThreadPool` 實例儲存發送者，如右邊範例所示。
 - 現在結構體 `Job` 還不會儲存任何東西，但是它最終會是傳送給通道的型別。
 - 在 `ThreadPool::new` 中，建立了一個新的通道並讓執行緒池儲存發送者。這能成功編譯，但還是會有些警告。

```
// --省略--
use std::{sync::mpsc, thread};

pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: mpsc::Sender<Job>,
}

struct Job;

impl ThreadPool {
    // --省略--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id));
        }

        ThreadPool { workers, sender }
    }
    // --省略--
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過通道傳遞請求給執行緒
 - 嘗試在執行緒池建立通道時，將接收者傳給每個 Worker。
知道想在 Worker 產生的執行緒中使用接收者，所以得在閉包中參考 receiver 參數。不過右邊範例的程式碼還不能編譯過。
 - 做了一些小小卻直觀的改變：將接收者傳給 Worker::new，然後在閉包中使用它。

```
impl ThreadPool {  
    // --省略--  
    pub fn new(size: usize) -> ThreadPool {  
        assert!(size > 0);  
  
        let (sender, receiver) = mpsc::channel();  
  
        let mut workers = Vec::with_capacity(size);  
  
        for id in 0..size {  
            workers.push(Worker::new(id, receiver));  
        }  
  
        ThreadPool { workers, sender }  
    }  
    // --省略--  
}  
  
// --省略--  
  
impl Worker {  
    fn new(id: usize, receiver: mpsc::Receiver<Job>) -> Worker {  
        let thread = thread::spawn(|| {  
            receiver;  
        });  
  
        Worker { id, thread }  
    }  
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過通道傳遞請求給執行緒
- 當檢查此程式碼時，會得到以下錯誤：

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
error[E0382]: use of moved value: `receiver`
  --> src/lib.rs:26:42
21 |         let (sender, receiver) = mpsc::channel();
   |         ----- move occurs because `receiver` has type `std::sync::mpsc::Receiver<Job>`,
   |         which does not implement the `Copy` trait
...
26 |         workers.push(Worker::new(id, receiver));
   |                                ^^^^^^^^^ value moved here, in previous iteration of loop
```

- 程式碼嘗試將 `receiver` 傳給數個 `Worker` 實例。回憶之前的話，就知道這不會成功：**Rust 提供的通道實作是多重生產者、單一消費者**。這意味著不能只是clone接收者來修正此程式碼。也不想重複傳送一個訊息給多重消費者，想要的是由數個工作者建立的訊息列表，然後每個訊息只會被處理一次。
- 除此之外，從通道佇列取得任務會需要可變的 `receiver`，所以執行緒需要有個安全的方式來共享並修改 `receiver`。不然的話，可能會遇到競爭條件。

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過通道傳遞請求給執行緒
 - 回想一下之前討論到的執行緒安全智慧指標：要在多重執行緒共享所有權並允許執行緒改變數值的話，需要使用 `Arc<Mutex<T>>`。
 - `Arc` 型別能讓數個工作者能擁有接收端，而 `Mutex` 能確保同時間只有一個工作者能獲取任務。
 - 右邊範例顯示了需要作出的改變：

```
use std::{
    sync::{mpsc, Arc, Mutex},
    thread,
};
// --省略--

impl ThreadPool {
    // --省略--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool { workers, sender }
    }

    // --省略--
}

// --省略--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --省略--
    }
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 透過通道傳遞請求給執行緒
 - 在 `ThreadPool::new` 中，將接收者放入 `Arc` 與 `Mutex` 之中。對於每個新的工作者，會 `clone` `Arc` 來增加參考計數，讓工作者可以共享接收者的所有權。
 - 有了這些改變，程式碼就能編譯了！就快完成了！

```
use std::{
    sync::{mpsc, Arc, Mutex},
    thread,
};
// --省略--

impl ThreadPool {
    // --省略--
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool { workers, sender }
    }

    // --省略--
}

// --省略--

impl Worker {
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {
        // --省略--
    }
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 實作 `execute` 方法

- 最後來對 `ThreadPool` 實作 `execute` 方法吧。還會將 `Job` 的型別從結構體改為特徵物件的型別別名，這會儲存 `execute` 收到的閉包型別。如同在「透過型別別名建立型別同義詞」段落所介紹的，型別別名能將很長的型別變短一些以便使用，如右上範例所示：`(src/lib.rs)`
- 在使用 `execute` 收到的閉包來建立新的 `Job` 實例之後，將該任務傳送至發送者。
- 對 `send` 呼叫 `unwrap` 來處理發送失敗的情況。舉例來說，這可能會發生在當停止所有執行緒時，這意味著接收端不再接收新的訊息。
- 不過目前還無法讓執行緒停止執行，只要執行緒池還在執行緒就會繼續執行。使用 `unwrap` 的原因是因為知道失敗不可能發生，但編譯器並不知情。

```
// --省略--  
  
type Job = Box<dyn FnOnce() + Send + 'static>;  
  
impl ThreadPool {  
    // --省略--  
  
    pub fn execute<F>(&self, f: F)  
    where  
        F: FnOnce() + Send + 'static,  
    {  
        let job = Box::new(f);  
  
        self.sender.send(job).unwrap();  
    }  
}  
  
// --省略--
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 實作 `execute` 方法

- 不過還沒結束呢！在工作者中，傳給 `thread::spawn` 的閉包仍然只有參考接收者。需要讓閉包一直循環，向接收者請求任務，並在取得任務時執行它。對 `Worker::new` 加上右上範例的程式碼：
- 在此首先對 `receiver` 呼叫 `lock` 以取得互斥鎖，然後呼叫 `unwrap` 讓任何錯誤都會恐慌。
- 如果互斥鎖處於污染(`poisoned`)狀態的話，該鎖可能就會失敗，這在其他執行緒持有鎖時，卻發生恐慌而沒有釋放鎖的話就可能發生。在這種情形，呼叫 `unwrap` 來讓此執行緒恐慌是正確的選擇。
- 也可以將 `unwrap` 改成 `expect` 來加上一些更有幫助的錯誤訊息。
- 如果得到互斥鎖的話，呼叫 `recv` 來從通道中取得 `Job`。最後的 `unwrap` 也繞過了任何錯誤，這在持有發送者的執行緒被關閉時就可能發生；就和如果接收端關閉時 `send` 方法就會回傳 `Err` 的情況類似。
- `recv` 的呼叫會阻擋執行緒，所以如果沒有任何任務的話，當前執行緒將等待直到下一個任務出現為止。
- `Mutex<T>` 確保同時間只會有一個 `Worker` 執行緒嘗試取得任務。

```
// --省略--  
  
impl Worker {  
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {  
        let thread = thread::spawn(move || loop {  
            let job = receiver.lock().unwrap().recv().unwrap();  
  
            println!("Worker {id} got a job; executing.");  
  
            job();  
        });  
  
        Worker { id, thread }  
    }  
}
```

建立多執行緒網頁伺服器

將單一執行緒伺服器轉換為多執行緒伺服器

- 實作 execute 方法

- 執行緒池終於可以運作了！ cargo run 然後下達一些請求吧：

成功了！現在有個執行緒池能非同步地處理連線。

產生的執行緒不超過四條，所以如果伺服器收到大量請求時，系統就不會超載。如果下達 `/sleep` 的請求，伺服器會有其他執行緒來處理其他請求並執行它們。

注意：如果在數個瀏覽器視窗同時打開 `/sleep`，它們可能會彼此間隔 5 秒鐘來讀取。這是因為有些網頁瀏覽器會對多個相同請求的實例做快取。這項限制不是網頁伺服器造成的。

```
$ cargo run
  Compiling hello v0.1.0 (file:///projects/hello)
warning: field is never read: `workers`
--> src/lib.rs:7:5
7 |         workers: Vec<Worker>,
  |         ^^^^^^^^^^^^^^^^^^^^^
= note: `#[warn(dead_code)]` on by default

warning: field is never read: `id`
--> src/lib.rs:48:5
48 |         id: usize,
    |         ^^^^^^^^

warning: field is never read: `thread`
--> src/lib.rs:49:5
49 |         thread: thread::JoinHandle<()>,
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

warning: `hello` (lib) generated 3 warnings
    Finished dev [unoptimized + debuginfo] target(s) in 1.40s
    Running `target/debug/hello`
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
Worker 1 got a job; executing.
Worker 3 got a job; executing.
Worker 0 got a job; executing.
Worker 2 got a job; executing.
```

建立多執行緒網頁伺服器

正常關機與清理

- 之前範例的程式碼能如所預期地使用執行緒池來同時回應多重請求。
- 有看到些警告說 `workers`、`id` 與 `thread` 欄位沒有被直接使用，這提醒尚未清理所有內容。當使用比較不優雅的 `ctrl-c` 方式來中斷主執行緒時，所有其他執行緒也會立即停止，不管它們是否正在處理請求。
- 接著要實作 `Drop` 特徵來對池中的每個執行緒呼叫 `join`，讓它們能在關閉前把任務處理完畢。然後要實作個方式來告訴執行緒它們該停止接收新的請求並關閉。
- 為了觀察此程式碼的實際運作，會修改伺服器讓它在正常關機(`graceful shutdown`)前，只接收兩個請求。

建立多執行緒網頁伺服器

正常關機與清理

- 對 ThreadPool 實作 Drop 特徵
 - 先對執行緒池實作Drop。當池被釋放時，執行緒都該加入(join)回來以確保它們有完成它們的工作。右上範例為實作 Drop 的第一次嘗試，不過此程式碼還無法執行：
 - 首先走訪執行緒池中的每個 workers。對此使用&mut因為self是個可變參考，而且也需要能夠改變 worker。對每個工作者印出訊息來說明此工作者正要關閉，然後對工作者的執行緒呼叫 join。
 - 如果 join 的呼叫失敗的話，使用 unwrap 來讓 Rust 恐慌使其變成較不正常的關機方式。

```
impl Drop for ThreadPool {  
    fn drop(&mut self) {  
        for worker in &mut self.workers {  
            println!("Shutting down worker {}", worker.id);  
  
            worker.thread.join().unwrap();  
        }  
    }  
}
```

建立多執行緒網頁伺服器

正常關機與清理

- 對 ThreadPool 實作 Drop 特徵
 - 以下是當編譯此程式碼時產生的錯誤：

```
$ cargo check
  Checking hello v0.1.0 (file:///projects/hello)
error[E0507]: cannot move out of `worker.thread` which is behind a mutable reference
--> src/lib.rs:52:13

52 |         worker.thread.join().unwrap();
   |         ^^^^^^^^^^^^^^^^^^ ----- `worker.thread` moved due to this method call
   |         |
   |         move occurs because `worker.thread` has type `JoinHandle<()>`, which does not implement the
   |         `Copy` trait
note: this function takes ownership of the receiver `self`, which moves `worker.thread`
```

- 錯誤告訴無法呼叫 join，因為只有每個 worker 的可變借用，而 join 會取走其引數的所有權。要解決此問題，需要將 thread 中的執行緒移出 Worker 實例，讓 join 可以消耗該執行緒。
- 在之前範例做過這樣的事，如果 Worker 改持有 Option<thread::JoinHandle<()>> 的話，可以對 Option 呼叫 take 方法來移動 Some 變體中的數值，並在原處留下 None 變體。
- 換句話說，thread 中有 Some 變體的話就代表 Worker 正在執行，而當清理 Worker 時，會將 Some 換成 None 來讓 Worker 沒有任何執行緒可以執行。

建立多執行緒網頁伺服器

正常關機與清理

- 對 ThreadPool 實作 Drop 特徵
 - 所以想要更新 Worker 的定義如以下所示：
 - 現在再看看編譯器的結果中還有哪些地方需要修改。檢查此程式碼，會得到兩個錯誤：

```
struct Worker {  
    id: usize,  
    thread: Option<thread::JoinHandle<()>>,  
}
```

```
$ cargo check  
    Checking hello v0.1.0 (file:///projects/hello)  
error[E0599]: no method named `join` found for enum `Option` in the current scope  
--> src/lib.rs:52:27  
52 |         worker.thread.join().unwrap();  
   |                        ^^^^^ method not found in `Option<JoinHandle<()>>`  
  
note: the method `join` exists on the type `JoinHandle<()>`  
help: consider using `Option::expect` to unwrap the `JoinHandle<()>` value, panicking if the value is an `Option::None`  
52 |         worker.thread.expect("REASON").join().unwrap();  
   |                        ++++++
```

```
error[E0308]: mismatched types  
--> src/lib.rs:72:22  
72 |         Worker { id, thread }  
   |                   ^^^^^^ expected enum `Option`, found struct `JoinHandle`  
  
= note: expected enum `Option<JoinHandle<()>>`  
        found struct `JoinHandle<_>`  
help: try wrapping the expression in `Some`  
72 |         Worker { id, thread: Some(thread) }  
   |                             ++++++ +  
  
error: aborting due to 2 previous errors
```

建立多執行緒網頁伺服器

正常關機與清理

- 對 ThreadPool 實作 Drop 特徵
 - 來修復第二個錯誤，這指向程式碼中 Worker::new 的結尾。當建立新的 Worker，需要將 thread 的數值封裝到 Some。請作出以下改變來修正程式碼：

```
impl Worker {  
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {  
        // --省略--  
  
        Worker {  
            id,  
            thread: Some(thread),  
        }  
    }  
}
```

建立多執行緒網頁伺服器

正常關機與清理

- 對 ThreadPool 實作 Drop 特徵
 - 而第一個錯誤則位在 Drop 的實作中。剛剛有提到打算對 Option 呼叫 take 來將 thread 移出 worker。

所以以下改變就能修正：

```
impl Drop for ThreadPool {  
    fn drop(&mut self) {  
        for worker in &mut self.workers {  
            println!("Shutting down worker {}", worker.id);  
  
            if let Some(thread) = worker.thread.take() {  
                thread.join().unwrap();  
            }  
        }  
    }  
}
```

- 如同之前所討論的，Option 的 take 方法會取走 Some 變體的數值並在原地留下 None。
- 使用 if let 來解構 Some 並取得執行緒，然後對執行緒呼叫 join。
- 如果工作者的執行緒已經是 None，就知道該該工作者已經清理其執行緒了，所以沒有必要再處理。

建立多執行緒網頁伺服器

正常關機與清理

- 對執行緒發送停止接收任務的信號
 - 有了以上的改變，程式碼就能成功編譯且沒有任何警告。但壞消息是此程式碼並沒有如所預期地運作。關鍵邏輯位於 **Worker** 實例中執行緒執行的閉包，現在雖然有呼叫 **join**，但這無法關閉執行緒，因為它們會一直 **loop** 來尋找任務執行。如果嘗試以目前的 **drop** 實作釋放 **ThreadPool** 的話，主執行緒會被阻擋，一直等待第一個執行緒處理完成。
- 要修正此問題，要修改 **ThreadPool drop** 的實作以及 **Worker** 內的一些程式碼。

建立多執行緒網頁伺服器

正常關機與清理

- 對執行緒發送停止接收任務的信號
 - 首先先將 ThreadPool drop 的實作改成在執行緒完成前就**顯式釋放 sender**。右邊範例展示了 ThreadPool 顯式釋放 sender。使用處理執行緒時一樣的 Option 與 take 技巧來將 sender 移出 ThreadPool：
 - 釋放 sender 會關閉通道，也就代表沒有任何訊息會再被傳送。工作者在無限迴圈呼叫的 recv 會回傳錯誤。

```
pub struct ThreadPool {
    workers: Vec<Worker>,
    sender: Option<mpsc::Sender<Job>>,
}
// --省略--
impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        // --省略--

        ThreadPool {
            workers,
            sender: Some(sender),
        }
    }

    pub fn execute<F>(&self, f: F)
    where
        F: FnOnce() + Send + 'static,
    {
        let job = Box::new(f);

        self.sender.as_ref().unwrap().send(job).unwrap();
    }
}

impl Drop for ThreadPool {
    fn drop(&mut self) {
        drop(self.sender.take());

        for worker in &mut self.workers {
            println!("Shutting down worker {}", worker.id);

            if let Some(thread) = worker.thread.take() {
                thread.join().unwrap();
            }
        }
    }
}
```

建立多執行緒網頁伺服器

正常關機與清理

- 對執行緒發送停止接收任務的信號
 - 在右上範例中，改變 Worker 的迴圈來處理該狀況並正常退出迴圈，也就是說 ThreadPool drop 的實作呼叫 join 時，執行緒就會工作完成。
 - 要實際看到此程式碼的運作情形，修改main來在正常關閉伺服器前，只接收兩個請求，如底下範例所示：

```
impl Worker {  
    fn new(id: usize, receiver: Arc<Mutex<mpsc::Receiver<Job>>>) -> Worker {  
        let thread = thread::spawn(move || loop {  
            let message = receiver.lock().unwrap().recv();  
  
            match message {  
                Ok(job) => {  
                    println!("Worker {id} got a job; executing.");  
  
                    job();  
                }  
                Err(_) => {  
                    println!("Worker {id} disconnected; shutting down.");  
                    break;  
                }  
            }  
        });  
  
        Worker {  
            id,  
            thread: Some(thread),  
        }  
    }  
}
```

```
fn main() {  
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();  
    let pool = ThreadPool::new(4);  
  
    for stream in listener.incoming().take(2) {  
        let stream = stream.unwrap();  
  
        pool.execute(|| {  
            handle_connection(stream);  
        });  
    }  
  
    println!("Shutting down.");  
}
```

建立多執行緒網頁伺服器

正常關機與清理

- 對執行緒發送停止接收任務的信號
- 在真實世界中的網頁伺服器當然不會只處理兩個請求就關機。此程式碼只是用來說明正常關機與清理的運作流程。take 方法是由 Iterator 特徵所定義且限制該疊代最多只會**取得前兩項**。
- ThreadPool 會在 main 結束時離開作用域，然後 drop 的實作就會執行。
- 使用 cargo run 開啟伺服器，並下達三個請求。第三個請求應該會出現錯誤，而在終端機中應該會看到類似以下的輸出：

```
fn main() {  
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();  
    let pool = ThreadPool::new(4);  
  
    for stream in listener.incoming().take(2) {  
        let stream = stream.unwrap();  
  
        pool.execute(|| {  
            handle_connection(stream);  
        });  
    }  
  
    println!("Shutting down.");  
}
```

```
$ cargo run  
Compiling hello v0.1.0 (file:///projects/hello)  
Finished dev [unoptimized + debuginfo] target(s) in 1.0s  
Running `target/debug/hello`  
Worker 0 got a job; executing.  
Shutting down.  
Shutting down worker 0  
Worker 3 got a job; executing.  
Worker 1 disconnected; shutting down.  
Worker 2 disconnected; shutting down.  
Worker 3 disconnected; shutting down.  
Worker 0 disconnected; shutting down.  
Shutting down worker 1  
Shutting down worker 2  
Shutting down worker 3
```

建立多執行緒網頁伺服器

正常關機與清理

- 對執行緒發送停止接收任務的信號
 - 可能會看到不同順序的工作者與訊息輸出。
 - 可以從訊息中看到此程式碼如何執行的，工作者 0 與 3 獲得前兩個請求。在第二個請求之後，伺服器會停止接受連線。然後在工作者 3 開始工作之前，ThreadPool 的 Drop 實作就會執行。
 - 釋放 sender 會將所有工作者斷線並告訴它們關閉。
 - 每個工作者在斷線時都印出訊息，然後執行緒池會呼叫 join 來等待每個工作者的執行緒完成。

```
$ cargo run
  Compiling hello v0.1.0 (file:///projects/hello)
  Finished dev [unoptimized + debuginfo] target(s) in 1.0s
  Running `target/debug/hello`
Worker 0 got a job; executing.
Shutting down.
Shutting down worker 0
Worker 3 got a job; executing.
Worker 1 disconnected; shutting down.
Worker 2 disconnected; shutting down.
Worker 3 disconnected; shutting down.
Worker 0 disconnected; shutting down.
Shutting down worker 1
Shutting down worker 2
Shutting down worker 3
```


建立多執行緒網頁伺服器

正常關機與清理

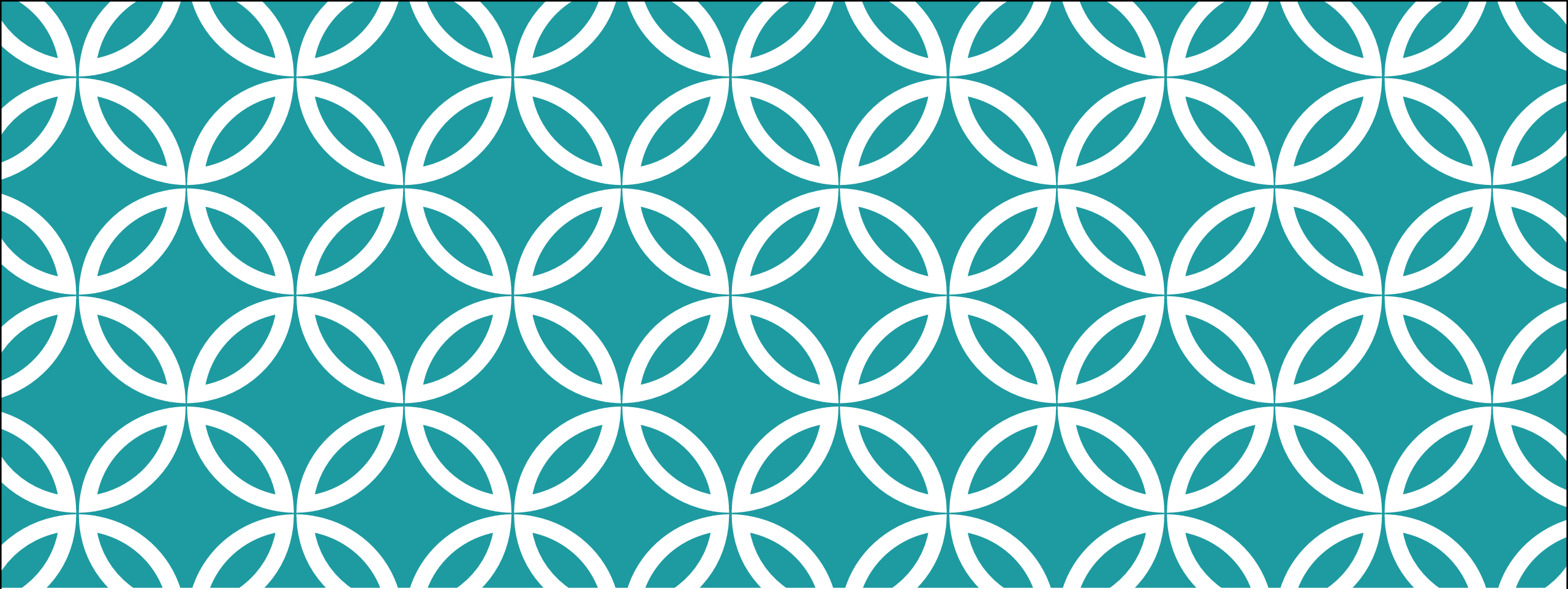
- 對執行緒發送停止接收任務的信號
 - 此特定執行方式中有個有趣的地方值得注意：在 `ThreadPool` 釋放 `sender` 然後任何工作者收到錯誤之前，嘗試將工作者 0 加入回來。工作者 0 尚未從 `recv` 收到錯誤，所以主執行緒會被擋住並等待工作者 0 完成。
 - 同一時間，工作者 3 收到一份工作但所有執行緒都收到錯誤。
 - 當工作者 0 完成時，主執行緒會等待剩下的工作者完成任務。屆時，它們都會退出它們的迴圈並能夠關閉。
 - 專案完成了，有個基礎的網頁瀏覽器，其使用執行緒池來做非同步回應。能夠對伺服器正常關機，並清理池中所有的執行緒。

```
$ cargo run
  Compiling hello v0.1.0 (file:///projects/hello)
  Finished dev [unoptimized + debuginfo] target(s) in 1.0s
  Running `target/debug/hello`
Worker 0 got a job; executing.
Shutting down.
Shutting down worker 0
Worker 3 got a job; executing.
Worker 1 disconnected; shutting down.
Worker 2 disconnected; shutting down.
Worker 3 disconnected; shutting down.
Worker 0 disconnected; shutting down.
Shutting down worker 1
Shutting down worker 2
Shutting down worker 3
```

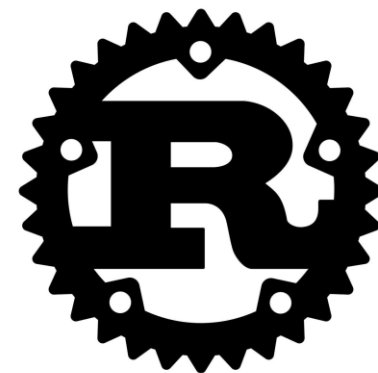
建立多執行緒網頁伺服器

正常關機與清理

- 對執行緒發送停止接收任務的信號
 - 還可以做更多事！如果想繼續改善此專案的話，以下是些不錯的點子：
 1. 對 `ThreadPool` 與其公開方法加上技術文件。
 2. 對函式庫功能加上測試。
 3. 將 `unwrap` 的呼叫改成更完善的錯誤處理。
 4. 使用 `ThreadPool` 來處理其他種類的任務，而不只是網頁請求。
 5. 在 `crates.io` 找到一個執行緒池 `crate`，並使用該 `crate` 實作類似的網頁伺服器。然後比較該 `crate` 與實作的執行緒池之間的 API 與穩固程度。



Actix-Web 、 Rocket與Yew介紹



Actix-Web、Rocket與Yew介紹

為什麼選擇 Actix Web ？

- 截至2024/6目前為止，Actix Web 在 GitHub 上所獲得的星星數是 20.5k，在所有 Rust Web 框架中屬於前段班，這也代表 Actix Web 是一個熱門並且有一定使用者的開源專案，以下也是為什麼選擇 Actix Web 的一些優點：

1. 高效能：

- Actix Web 的效能不只是在 Rust 開發圈屬於領頭羊，並且在眾多語言的 Web 框架中也是出了名的高效能。這意味著可以用較少的資源達到更多的請求處理。

2. 非同步支持：

- 利用 Rust 的非同步特性，Actix Web 能夠輕鬆地處理高並發的網頁應用。

3. Middleware 系統：

- 例如錯誤處理、日誌記錄或認證等功能都變得簡單且模組化。

Actix-Web、Rocket與Yew介紹

安裝 Actix Web

- 那麼就來建立一個新的 Rust 專案吧！這個專案命名為"alvin-code-generator"，新增完之後就移動到專案目錄中：

```
$ cargo new alvin-code-generator  
$ cd alvin-code-generator
```
- 接下來安裝方式可以直接在 Terminal 執行：

```
$ cargo add actix-web
```
- Cargo 就會安裝 Actix Web 最新的穩定版本到 Rust 專案中。或者，也可以在專案中根目錄下的 cargo.toml，並且在 [dependencies] 區塊下面加入以下文字，那個數字代表的是目前的版本：

```
[dependencies]  
actix-web = "4"
```
- 然後，執行 cargo build 或是 cargo run，讓 Cargo 安裝這個套件。

Actix-Web、Rocket與Yew介紹

建立第一個 Actix Web 應用

- 開始的第一步都先從建立一個簡單的 "Hello World" 開始吧！
- 找到 main.rs 後，並把裡面的程式碼全部替換成以下：

```
use actix_web::{get, App, HttpResponse, HttpServer, Responder};

#[get("/")]
async fn hello() -> impl Responder {
    HttpResponse::Ok().body("Hello, World!")
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .service(hello)
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

接下來在 Terminal 執行 `cargo run`。現在，可以打開瀏覽器，在 URL 路徑輸入 `http://127.0.0.1:8080`，沒意外的話應該就可以看到 "Hello, World!" 顯示在畫面上。

Actix-Web 、 **Rocket** 與 Yew 介紹

Rocket 是什麼？

- 簡單介紹一下 Rocket，Rocket 是一個基於 Rust 語言的一個 Web 框架，截止目前為止在 GitHub 上一共有 23.6k 的星星數。
- Rocket 的設計靈感來自於 Ruby on Rails、Flask 等知名的 Web 框架，並且在設計上有著類似的特性，並以 3 個核心概念來設計：
 1. 安全、正確，還有重視開發者體驗。
 2. 所有的請求都會由 Rocket 來處理型別。
 3. 不強制使用 Rocket 的函式庫，可以讓開發者自由選擇。
- Rocket 的文件寫的非常的完整，算是蠻適合新手入門的一個框架。

Actix-Web 、 **Rocket** 與 Yew 介紹

從 Hello World 開始

- 那麼就從一個簡單的 Hello World 開始，首先先建立一個 binary-based 的新專案：

```
$ cargo new --bin rocket-server
```

安裝 Rocket 與第一支程式

- 在 Rust 安裝套件非常容易，只要在 Cargo.toml 裡面 [dependencies] 的下方加入要安裝的套件就可以了，在這裡像這樣：

```
[dependencies]  
rocket = "0.5.0-rc.2"
```


Actix-Web 、 **Rocket** 與 Yew 介紹

安裝 Rocket 與第一支程式

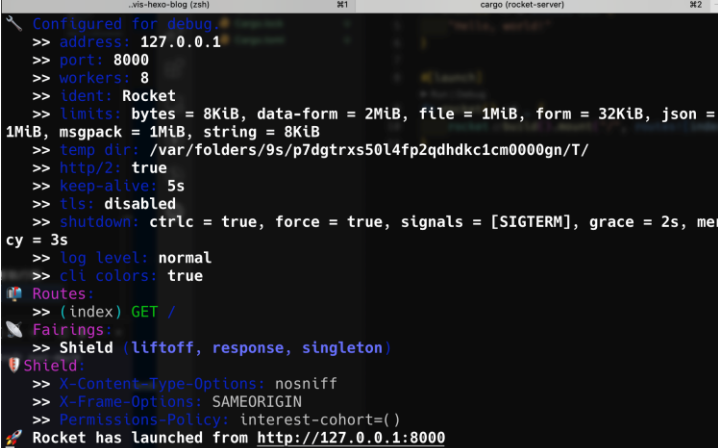
- 然後在 main.rs 修改一下 code：

```
#[macro_use] extern crate rocket;

#[get("/")]
fn index() -> &'static str {
    "Hello, world!"
}

#[launch]
fn rocket() -> _ {
    rocket::build().mount("/", routes![index])
}
```

- 接著在終端機輸入 `cargo run`，一開始會先下載 Rocket 的套件，然後會接著執行程式，終端機的畫面會是這樣：



```
Configured for debug.
>> address: 127.0.0.1
>> port: 8000
>> workers: 8
>> ident: Rocket
>> limits: bytes = 8KiB, data-form = 2MiB, file = 1MiB, form = 32KiB, json = 1MiB, msgpack = 1MiB, string = 8KiB
>> temp_dir: /var/folders/9s/p7dgtrxs50l4fp2qdhdkc1cm0000gn/T/
>> http2: true
>> keep-alive: 5s
>> tls: disabled
>> shutdown: ctrlc = true, force = true, signals = [SIGTERM], grace = 2s, mercy = 3s
>> log level: normal
>> cli colors: true
Routes:
>> (index) GET /
Fairings:
>> Shield (liftoff, response, singleton)
Shield:
>> X-Content-Type-Options: nosniff
>> X-Frame-Options: SAMEORIGIN
>> Permissions-Policy: interest-cohort=()
Rocket has launched from http://127.0.0.1:8000
```

Actix-Web 、 Rocket 與 Yew 介紹

安裝 Rocket 與第一支程式

- 都沒問題的話，可以按照提示連去網址，或去<http://localhost:8000>，可以看到瀏覽器顯示出Hello World 了。
- 以上就是使用 Rocket 建立一個簡單的 Hello World，跟之前用純 Rust 相比是不是感受到方便呢？
- 講解一下剛剛運作的 code，首先先引入 Rocket 的套件：
- 然後定義一個 index 的函式，並且回傳一個 &'static str，這個 &'static str 代表的是一個字串的 reference，而且這個 reference 的生命週期是 'static，也就是整個程式的生命週期：

```
#[macro_use] extern crate rocket;
```

```
#[get("/")]  
fn index() -> &'static str {  
    "Hello, world!"  
}
```

Actix-Web 、 **Rocket** 與 Yew 介紹

安裝 Rocket 與第一支程式

- 然後在 `#[launch]` 的函式裡面，使用 `rocket::build()` 來建立一個 Rocket 的實例，然後使用 `mount` 來將 `index` 函式 `mount` 到 `/` 的路徑上，最後回傳這個 Rocket 實例：

```
#[launch]
fn rocket() -> _ {
    rocket::build().mount("/", routes![index])
}
```

- Rocket 在這裡也提供了另一種啟動 server 的寫法：

```
#[rocket::main]
async fn main() -> Result<(), rocket::Error> {
    let _rocket = rocket::build()
        .mount("/", routes![index])
        .launch()
        .await?;

    Ok(())
}
```

- 兩者都可以運行，但是 `#[launch]` 的寫法比較簡潔，不過還是有些許的差異，例如：`#[launch]` 會自動建立一個 `main` 函式，而如果使用 `#[rocket::main]` 的話，開發者自己處理的部分可能就比較多，端看個人喜好。

Actix-Web 、 Rocket 與 Yew 介紹

JSON response

- 前面都只是回傳一個字串，但是如果要回傳一個 JSON 的話，該怎麼做呢？
- 這裡可以使用 Serde 這個框架來幫忙處理 JSON，首先要先在 Cargo.toml 裡面加入，並且修改一下：

```
[dependencies]
rocket = { version = "0.5.0-rc.2", features = ["json"] }
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

- 接著在 main.rs 裡面修改一下：

```
#[macro_use]
extern crate rocket;

use rocket::serde::json::{json, Json, Value};

#[get("/")]
fn index() -> Value {
    json!({ "name": "Bucky", "age": 18, "favorite_games": ["splatoon 3", "zelda"] })
}

#[launch]
fn rocket() -> _ {
    let routes = routes![index];
    rocket::build().mount("/", routes)
}
```

- 接著一樣重新執行，然後瀏覽器重新整理，就可以看到回傳的 JSON 了。

Actix-Web 、 Rocket與Yew介紹

WebAssembly

- WebAssembly是一種可以在瀏覽器中執行的程式語言，它可以在瀏覽器中執行非常複雜的程式，而且它的執行速度也非常快，所以可以在瀏覽器中執行一些比較複雜的運算，而不會影響到瀏覽器的體驗。

Yew

- Yew 是一個可以在 Rust 中使用 WebAssembly 的框架，它可以在 Rust 中撰寫前端的程式，並且可以編譯成 WebAssembly在瀏覽器中執行。

Actix-Web 、 Rocket與Yew介紹

安裝 Yew

- 那麼要如何使用 Yew 來開發前端的程式呢？按照官網的指示，要先安裝以下的東西：

- **WebAssembly** : `$ rustup target add wasm32-unknown-unknown`

- **Trunk** : `$ cargo install trunk`

開新專案

```
$ cargo new yew-demo
```

- 然後在 Cargo.toml 中加入以下的內容：

```
[dependencies]
yew = "0.19"
```

- 到這邊就可以開始撰寫程式了！

Actix-Web 、Rocket與Yew介紹

第一個Yew程式

- 先在 src/main.rs 中加入以下的內容：
- 然後在根目錄下新增一個 index.html：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Yew App</title>
  </head>
  <body>
  </body>
</html>
```

```
use yew::prelude::*;

enum Msg {
    AddOne,
}

struct Model {
    value: i64,
}

impl Component for Model {
    type Message = Msg;
    type Properties = ();

    fn create(_ctx: &Context<Self>) -> Self {
        Self {
            value: 0,
        }
    }

    fn update(&mut self, _ctx: &Context<Self>, msg: Self::Message) -> bool {
        match msg {
            Msg::AddOne => {
                self.value += 1;
                // the value has changed so we need to
                // re-render for it to appear on the page
                true
            }
        }
    }

    fn view(&self, ctx: &Context<Self>) -> Html {
        // This gives us a component's "Scope" which allows us to send messages, etc to the component.
        let link = ctx.link();
        html! {
            <div>
                <button onclick={link.callback(|_| Msg::AddOne)}>{ "+1" }</button>
                <p>{ self.value }</p>
            </div>
        }
    }
}

fn main() {
    yew::start_app::<Model>();
}
```

- 然後執行 **trunk serve**，就可以在瀏覽器中看到網頁了！

Actix-Web 、Rocket與Yew介紹

第一個Yew程式

- 不過目前只能讓數字加 1，再加另一個按鈕可以 - 1 看起來比較完整一點：
- 如果要順便加上一些 CSS 的話，可以在在根目錄新增一個 style.css，然後在 index.html 中加入：

```
<link data-trunk href="style.css" rel="css">
```

```
enum Msg {
    AddOne,
    MinusOne,
}

impl Component for Model {
    // 省略..
    fn update(&mut self, _ctx: &Context<Self>, msg: Self::Message) -> bool {
        match msg {
            Msg::AddOne => {
                self.value += 1;
                true
            }
            Msg::MinusOne => {
                if (self.value > 0) {
                    self.value -= 1;
                }
                true
            }
        }
    }

    fn view(&self, ctx: &Context<Self>) -> Html {
        let link = ctx.link();
        html! {
            <div class="content">
                <button onclick={link.callback(|_| Msg::AddOne)} class="btn">{ "+1" }</button>
                <p class="result">{ self.value }</p>
                <button onclick={link.callback(|_| Msg::MinusOne)} class="btn">{ "-1" }</button>
            </div>
        }
    }
}
```


Actix-Web 、Rocket與Yew介紹

第一個Yew程式

- 這樣就完成一個用 Yew 寫的 App 了！

