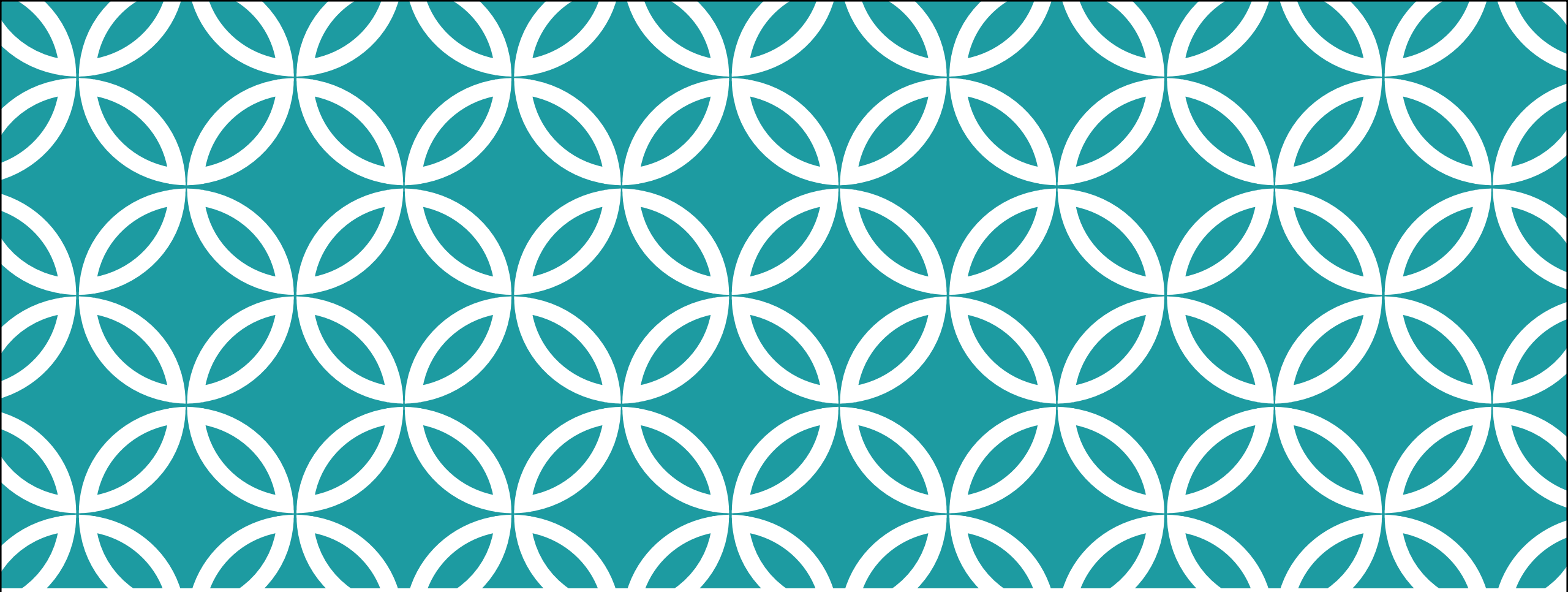


函式(Functions)

函式(Functions)

已見過這個最重要的函式 - main 函式是許多程式的入口點。

fn 關鍵字能宣告新的函式。

Rust 使用**snake case **式作為函式與變數名稱的慣例風格:

- **所有的字母都是小寫**
- **並用底線區隔單字**

函式(Functions)

基本函式

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

    another_function();
}

fn another_function() {
    println!("另一支函式。");
}
```

```
> cargo run
   Compiling my-project v0.1.0 (/home/runner/test-6)
    Finished dev [unoptimized + debuginfo] target(s) in 5.46s
     Running `target/debug/my-project`
Hello, world!
另一支函式。
```

- 在 Rust 中定義函式是先從 fn 開始，再加上函式名稱和一組括號，大括號告訴編譯器函式本體的開始與結束位置。
- 可以輸入函式的名稱並加上括號來呼叫任何定義過的函式。因為 another_function 已經在程式中定義了，就可以在 main 函式中呼叫。注意是在原始碼中的 main 函式之後定義 another_function 的，當然也可以把它定義在前面。Rust 不在乎函式是在哪裡定義的，只需要知道它定義在作用域的某處，且能被呼叫者看到就好。

函式(Functions)

參數(parameters)

- 函式可以定義成擁有參數(parameters)的，這是**函式簽名(signatures)**中特殊的變數，傳遞的數值則會叫做引數(arguments)。
  - 函式定義時才叫參數
  - 傳遞數值時叫做引數
- 以下為函式加上參數的範例：

```rust
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("x 的數值為：{x}");
}
```

```
> cargo run
    Blocking waiting for file lock on build directory
   Compiling my-project v0.1.0 (/home/runner/test-6)
    Finished dev [unoptimized + debuginfo] target(s) in 4.26s
     Running `target/debug/my-project`
x 的數值為： 5
```

  - 在函式簽名中，必須宣告每個參數的型別，這是 Rust 刻意做下的設計決定：在函式定義中要求型別詮釋，代表編譯器幾乎不需要在其他地方再提供資訊才能知道要使用什麼型別。而且如果編譯器能知道函式預期的型別的話，它還能夠給予更有幫助的錯誤訊息。

函式(Functions)

參數(parameters)

- 以下為函式加上參數的範例：如果要定義函式擁有數個參數時，會用逗號區隔開來。

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("測量值爲：{value}{unit_label}");
}
```

- 此範例建立了一個有兩個參數的函式 print_labeled_measurement，第一個參數叫做 value 而型別爲 i32，第二個參數叫做 unit_label 而型別爲 char。接著函式會印出包含 value 與 unit_label 的文字。

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/functions`
測量值爲：5h
```

函式(Functions)

函式回傳值

- 函式可以回傳數值給呼叫它們的程式碼，不會為回傳值命名，但必須用箭頭(->)來宣告型別。在 Rust 中，回傳值其實就是函式本體最後一行的表達式。可以用 return 關鍵字加上一個數值來提早回傳函式，**但多數函式都能用最後一行的表達式作為數值回傳**。以下是一個有回傳數值的函式範例：

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("x 的數值為：{x}");
}
```

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/functions`
x 的數值為：5
```

- 在 five 函式中沒有任何函式呼叫、巨集甚至是 let 陳述式，只有一個 5。這在 Rust 中完全是合理的函式。請注意到函式的回傳型別也有指明，就是 -> i32。嘗試執行此程式的話，輸出結果就會如右上：

函式(Functions)

```rust
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("x 的數值為：{x}");
}
```

函式回傳值

- five 中的 5 就是函式的回傳值，這就是為何回傳型別是 i32。進一步研究細節，這邊有兩個重要的地方：首先這行 let x = five(); 顯示了用函式的回傳值作為變數的初始值。因為函式 five 回傳 5，所以這行和以下程式碼相同：

```rust
let x = 5;
```

- 再來，five 函式沒有參數但有定義回傳值的型別。所以函式本體只需有一個 5 就好，不需加上分號，這樣就能當做表達式回傳想要的數值。再看另一個例子：

```rust
fn main() {
    let x = plus_one(5);

    println!("x 的數值為：{x}");
}

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

- **執行此程式會顯示 x 的數值為：6，但如果在最後一行 x + 1 加上分號的話，就會將它從表達式變為陳述式，就會得到錯誤**：

函式(Functions)

```rust
fn main() {
    let x = plus_one(5);

    println!("x 的數值為：{x}");
}

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

函式回傳值

- 錯誤訊息 mismatched types 就告訴此程式碼的核心問題。plus_one 的函式定義說它會回傳 i32 但是**陳述式不會回傳任何數值**。用單元型別()表示不會回傳任何值。因此沒有任何值被回傳，這和函式定義相牴觸，最後產生錯誤。在此輸出結果，Rust 提供了一道訊息來協助解決問題：**建議移除分號**，這樣就能修正錯誤：

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
error[E0308]: mismatched types
 --> src/main.rs:7:24
  |
7 | fn plus_one(x: i32) -> i32 {
  |    --------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
8 |     x + 1;
  |          - help: removing this semicolon

For more information about this error, try `rustc --explain E0308`.
error: could not compile `functions` due to previous error
```

函式(Functions)

使用函式進行基本算術運算

```
fn sub_s(a: i32, b: i32) -> i32 {
    a - b
}

fn display_sth(result: i32) {
    println!("{:?}", result);
}

fn main() {
    let result = sub_s(42, 7);
    display_sth(result);
}
```

所有權

所有權

什麼是所有權？

- 所有權是在 Rust 中用來**管理程式記憶體的一系列規則**。
- 所有程式都需要在執行時管理它們使用記憶體的方式。**有些語言會用垃圾回收機制(GC)，在程式執行時不斷尋找不再使用的記憶體；而有些程式，開發者必須親自配置和釋放記憶體。Rust 選擇了第三種方式：記憶體由所有權系統管理，且編譯器會在編譯時加上一些規則檢查。如果有地方違規的話，程式就無法編譯**。這些所有權的規則完全不會降低執行程式的速度。
- 當理解所有權時，就有一個穩健的基礎能夠理解那些使 Rust 獨特的功能。接下來將透過一些範例來學習所有權，會專注在一個非常常見的資料結構：**字串**。

所有權

堆疊(Stack)與堆積(Heap)

- 在許多程式語言中，通常不需要去想到堆疊與堆積。但在像是 Rust 這樣的系統程式語言，資料是存放於堆疊還是堆積就會有差，這會影響語言的行為也是得作出某些特定決策的理由。
- 堆疊與堆積都是提供程式碼在執行時能夠使用的記憶體部分，但組成的方式卻不一樣。**堆疊會按照它取得數值的順序依序存放它們，並以相反的順序移除數值。這通常稱為後進先出(last in, first out)**。以把堆疊想成是盤子，當要加入更多盤子，會將它們疊在最上面。如果要取走盤子的話，也是從最上方拿走。想要從底部或中間，插入或拿走盤子都是不可行的！當要**新增資料**時，會稱呼為**推入堆疊(pushing onto the stack)**，而**移除資料**則叫做**彈出堆疊(popping off the stack)**。**所有在堆疊上的資料都必須是已知固定大小(靜態)。在編譯時屬於未知或可能變更大小(動態)的資料必須儲存在堆積**。
- 堆積就比較沒有組織，當要將資料放入堆積，得要求一定大小的空間。**記憶體配置器(memory allocator)會找到一塊夠大的空位，標記為已佔用，然後回傳一個指標(pointer)，指著該位置的位址**。這樣的過程稱為在堆積上配置(allocating on the heap)或者有時直接簡稱為配置**(allocating)**就好(將數值放入堆疊不會被視為是在配置)。**因為指標是固定已知的大小，所以可以存在堆疊上。但當要存取實際資料時，就得去透過指標取得資料**。
- 可以想像成是一個餐廳。當進入餐廳時，會告訴服務員團體有多少人，他就會將團體帶到足夠人數的餐桌。如果團體有人晚到的話，可以直接詢問坐在哪而找到團體。

所有權

堆疊(Stack)與堆積(Heap)

- **將資料推入堆疊會比在堆積上配置還來的快**，因為配置器不需要去搜尋哪邊才能存入新資料，其位置永遠在堆疊最上方。相對的，堆積就需要比較多步驟，配置器必須先找到一個夠大的空位來儲存資料，然後作下紀錄為下次配置做準備。
- **在堆積上取得資料也比在堆疊上取得來得慢**，因為需要用追蹤指標才找的到。現代的處理器如果在記憶體間跳轉越少的話速度就越快。繼續用餐廳做比喻，想像伺服器就是在餐廳為數個餐桌點餐。最有效率的點餐方式就是依照餐桌順序輪流點餐。如果幫餐桌 A 點了餐之後跑到餐桌 B 點，又跑回到 A 然後又跑到 B 的話，可以想像這是個浪費時間的過程。同樣的道理，處理器在處理任務時，如果處理的資料相鄰很近(就如同存在堆疊)的話，當然比相鄰很遠(如同存在堆積)來得快。
- 當程式碼呼叫函式時，傳遞給函式的數值(可能包含指向堆積上資料的指標)與函式區域變數會被推入堆疊。當函式結束時，這些數值就會被彈出。
- 追蹤哪部分的程式碼用到了堆積上的哪些資料、最小化堆積上的重複資料、以及清除堆積上沒在使用的資料確保不會耗盡空間，這些問題都是所有權系統要處理的。一旦理解所有權後，通常就不再需要經常考慮堆疊與堆積的問題，**不過能理解所有權主要就是為了管理堆積有助於解釋為何它要這樣運作**。

所有權

所有權規則

- 首先，先看看所有權規則。當解釋說明時，請記得這些規則：

1. **Rust 中每個數值都有個擁有者(owner)。**
2. **同時間只能有一個擁有者。**
3. **當擁有者離開作用域時，數值就會被丟棄。**

所有權

變數作用域

- 現在既然已經知道了基本語法，接下來就不再將 fn main() { 寫進程式碼範例範例中。在參考時，請記得親自寫在 main 函式內。這樣一來，的範例可以更加簡潔，更加專注在細節而非樣板程式。
- 作為所有權的第一個範例，先來看變數的作用域(scope)。作用域是一些項目在程式內的有效範圍。假設有以下變數：

```
let s = "hello";
```

- 變數 s 是一個字串字面值(string literal)，而字串數值是寫死在程式內。此變數的有效範圍是從它宣告開始一直到當前作用域結束為止：

```
{                       // s 在此處無效，因為它還沒宣告
    let s = "hello";    // s 在此開始視為有效

    // 使用 s
}                       // 此作用域結束， s 不再有效
```

換句話說，這裡有兩個重要的時間點：

1. **當 s 進入作用域時，它是有效的。**
2. **它持續被視為有效直到它離開作用域為止。**

所有權

String 型別

- 要能夠解釋所有權規則，需要使用比「資料型別」還複雜的型別才行。之前提到的型別都是**已知固定大小且儲存在堆疊上的，在作用域結束時就會從堆疊中彈出**。而且如果其它部分程式碼需要在不同作用域使用相同數值的話，它們都能迅速簡單地透過複製產生新的單獨實例。但是想要觀察的是儲存在堆積上的資料，並研究 Rust 是如何知道要清理資料的。而 String 型別正是個絕佳範例。專注在 String 與所有權有關的部分。這些部分也適用於其他基本函式庫或自己定義的複雜資料型別。
- 已經看過字串字面值(string literals)，字串的數值是寫死在程式內的。字串字面值的確很方便，但它不可能完全適用於使用文字時的所有狀況。**其中一個原因是因為它是不可變的**，另一個原因是並非所有字串值在撰寫程式時就會知道。舉例來說，要是想要收集使用者的輸入並儲存它呢？對於這些情形，Rust 提供第二種字串型別(String)。此型別管理配置在堆積上的資料，所以可以儲存在編譯期間未知的一些文字。

所有權

String 型別

- 可以從字串字面值使用 from 函式來建立一個 String，如以右所示：

```
let s = String::from("hello");
```

- **雙冒號::可以將 from 函式置於 String 型別的命名空間(namespace)底下**，而不是取像是 string_from 這樣的名稱。將會在之後的「方法語法」討論這個語法，並在「參考模組項目的路徑」討論模組(modules)與命名空間。這種類型的字串是**可以被改變**的：

```
let mut s = String::from("hello");

s.push_str(", world!"); // push_str() 將字面值加到字串後面

println!("{}", s); // 這會印出 `hello, world!`
```

- 所以這邊有何差別呢？為何 String 是可變的，但字面值卻不行？兩者最主要的差別在於它們對待記憶體的方式。

所有權

記憶體與配置

- 以字串字面值來說，在編譯時就知道它的內容，所以可以寫死在最終執行檔內。這就是為何字串字面值非常迅速且高效。但這些特性均來自於字串字面值的**不可變性**。不幸的是無法將編譯時未知大小的文字，或是執行程式時大小可能會改變的文字等對應記憶體塞進執行檔中。
- 而對於 String 型別來說，為了要能夠**支援可變性**、改變文字長度大小，需要在堆積上配置一塊編譯時未知大小的記憶體來儲存這樣的內容，這代表：
  1. **記憶體配置器必須在執行時請求記憶體。**
  2. **不再需要這個 String 時，需要以某種方法將此記憶體還給配置器。**

所有權

記憶體與配置

- 當呼叫 String::from 時就等於完成**第一個部分**，它的實作會請求配置一塊它需要的記憶體。這邊大概和其他程式語言都一樣。
- 不過**第二部分**就不同了。在擁有垃圾回收機制(garbage collector, GC)的語言(Java)中，GC 會追蹤並清理不再使用的記憶體，所以不用去擔心這件事。沒有GC(C語言)的話，識別哪些記憶體不再使用並明確的呼叫程式碼釋放它們就是該做的責任了，就像請求取得它一樣。在以往的歷史可以看到要完成這件事是一項艱鉅的任務，**如果忘了，那麼就等於在浪費記憶體。如果釋放太早的話，則有可能會拿到無效的變數。要是釋放了兩次，那也會造成程式錯誤。必須準確無誤地配對一個 allocate 給剛好一個 free**。

所有權

記憶體與配置

- Rust 選擇了一條不同的道路：**當記憶體在擁有它的變數離開作用域時就會自動釋放**。以下是解釋作用域的範例，但使用的是 String 而不是原本的字串字面值：

```
{
    let s = String::from("hello"); // s 在此開始視為有效

    // 使用 s
}                                  // 此作用域結束
                                   // s 不再有效
```

注意：在 C++，這樣在項目生命週期結束時釋放資源的模式，有時被稱為資源取得即初始化 (Resource Acquisition Is Initialization, RAII)。如果已經用過 RAII 的模式，那麼應該就會很熟悉 Rust 的 drop 函式。

- 當 s 離開作用域時，就可以自然地將 String 所需要的記憶體釋放回配置器。**當變數離開作用域時，Rust 會呼叫一個特殊函式，此函式叫做 drop**。在這裡當時 String 的撰寫者就可以寫入釋放記憶體的程式碼。Rust 會在大括號結束時自動呼叫 drop。
- 這樣的模式對於 Rust 程式碼的編寫有很深遠的影響。雖然現在這樣看起來很簡單，但在更多複雜的情況下程式碼的行為可能會變得很難預測。像是當需要許多變數，所以得在堆積上配置它們的情況。

所有權

變數與資料互動的方式：移動(Move)

- 數個變數在 Rust 中可以有許多不同方式來與相同資料進行互動。看看使用整數的範例：

```
let x = 5;
let y = x;
```

- 大概可以猜到這做了什麼：「x 取得數值 5，然後拷貝(copy)了一份 x 的值給 y。」所以有兩個變數 x 與 y，而且都等於 5。這的確是所想的這樣，因為整數是已知且固定大小的簡單數值，所以這兩個數值 5 都會推入堆疊中。現在看看 String 的版本：

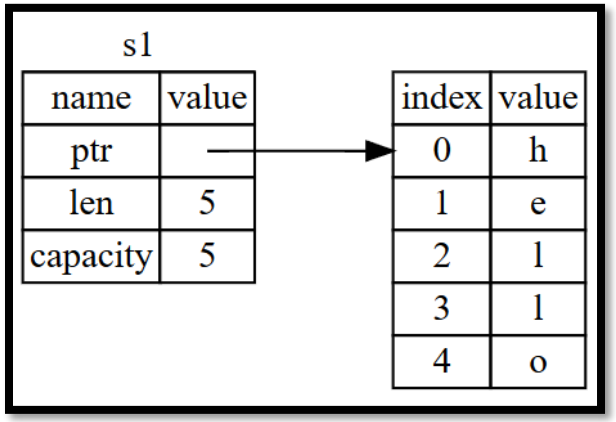
```
let s1 = String::from("hello");
let s2 = s1;
```

- 這和之前的程式碼非常相近，所以可能會認為它做的事也是一樣的：也就是第二行也會拿到一份 s1 拷貝的值給 s2。**但事實上卻不是這樣**。

所有權

變數與資料互動的方式：移動(Move)

- 請看看底下圖來瞭解 String 底下的架構到底長什麼樣子。一個 String 由三個部分組成，如圖中左側所示：一個指向儲存字串內容記憶體的指標、它的長度和它的容量。這些**資料是儲存在堆疊上的**，但圖右的**內容則是儲存在堆積上**。



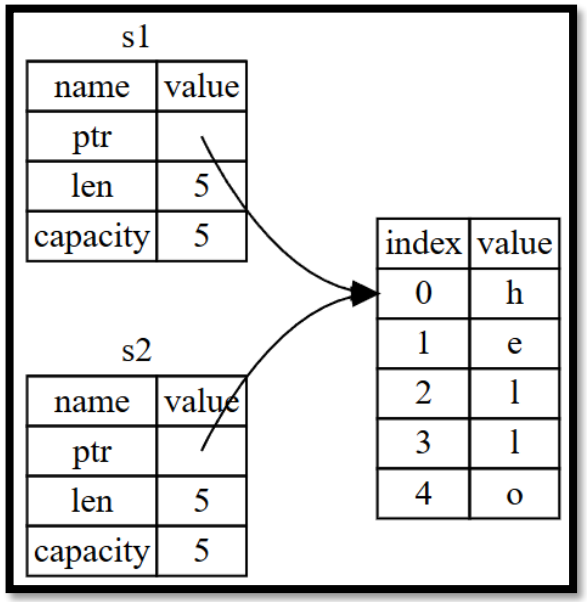
將數值 "hello" 賦值給 s1 的 String 記憶體結構

- **長度指的是目前所使用的 String 內容在記憶體以位元組為單位所佔用的大小。而容量則是 String 從配置器以位元組為單位取得的總記憶體量**。長度和容量是有差別的，但現在來說還不太重要，現在可以先忽略容量的問題。

所有權

變數與資料互動的方式：移動(Move)

- 當將 s1 賦值給 s2，String 的資料會被拷貝，不過拷貝的是堆疊上的指標、長度和容量。不會拷貝指標指向的堆積資料。資料以記憶體結構表示的方式會如下圖示表示：



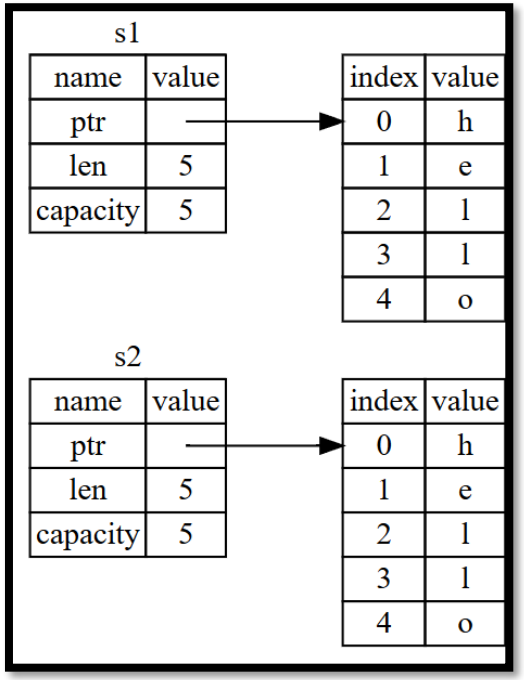
s2 擁有一份 s1 的指標、長度和容量的記憶體結構

所有權

變數與資料互動的方式：移動(Move)

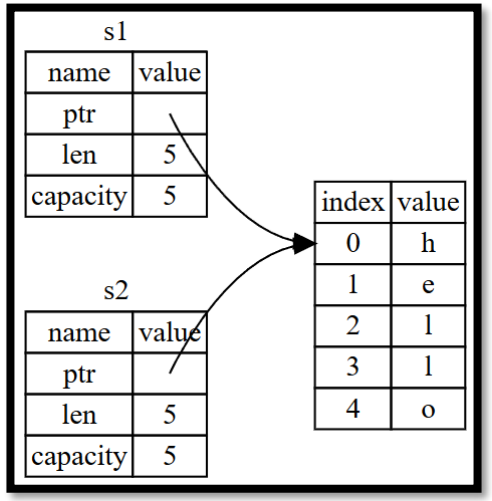
- 所以實際上的結構不會長的像下圖這樣，**如果 Rust 也會拷貝堆積資料的話**，才會看起來像這樣。

  如果 Rust 這麼做的話，s2 = s1 的動作花費會變得非常昂貴。當堆積上的資料非常龐大時，對執行時的性能影響是非常顯著的。



如果 Rust 也會拷貝堆積資料，s2 = s1 可能會長得樣子

所有權



變數與資料互動的方式：移動(Move)

- 稍早提到當變數離開作用域時，Rust 會自動呼叫 drop 函式並清理該變數在堆積上的資料。但圖示顯示兩個資料指標都指向相同位置，這會造成一個問題。**當 s2 與 s1 都離開作用域時，它們都會嘗試釋放相同的記憶體。這被稱呼為雙重釋放(double free)錯誤，也是之前提過的錯誤之一。釋放記憶體兩次可能會導致記憶體損壞，進而造成安全漏洞。**
- 為了保障記憶體安全，在此情況中 Rust 還會再做一件重要的事。在 let s2 = s1; 之後，Rust 就不再將 s1 視爲有效。因此當 s1 離開作用域時，Rust 不需要釋放任何東西。**請看看如果在 s2 建立之後繼續使用 s1 會發生什麼事，以下程式就執行不了：**

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

所有權

變數與資料互動的方式：移動(Move)

- 會得到像這樣的錯誤，因為 Rust 會防止使用無效的參考：

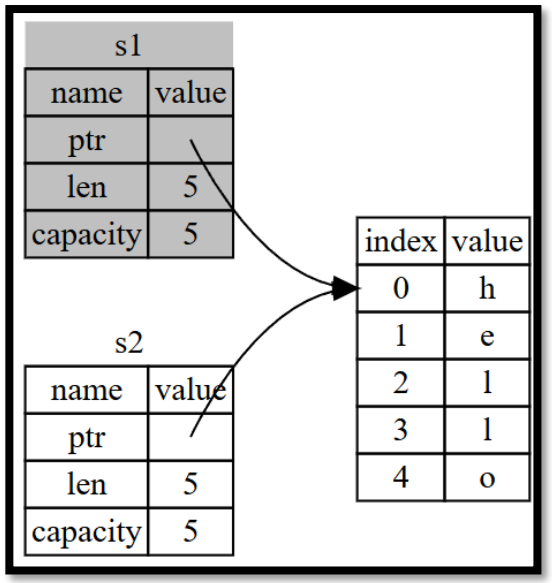
```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not implement the
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` which comes from

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` due to previous error
```

所有權

變數與資料互動的方式：移動(Move)

- 如果在其他語言聽過淺拷貝(shallow copy)和深拷貝(deep copy)這樣的詞，**拷貝指標、長度和容量而沒有拷貝實際內容這樣的概念應該就相近於淺拷貝**。但因為 Rust 同時又無效化第一個變數，不會叫此為淺拷貝，**而是稱此動作為移動(move)**。在此範例會稱 s1 被移動到 s2，所以實際上發生的事長得像底下圖示這樣：

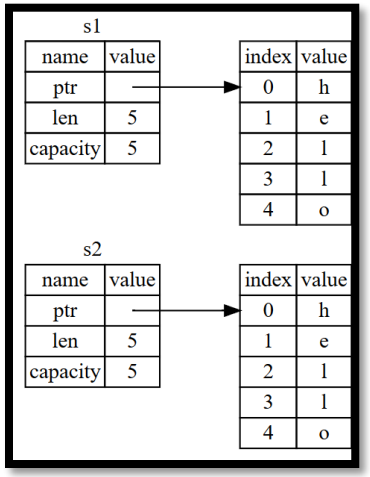


s1 無效後的記憶體結構

這樣就解決了問題！只有 s2 有效的話，當它離開作用域，就只有它會釋放記憶體，就完成所有動作了。

除此之外，這邊還表達了另一個設計決策：**Rust 永遠不會自動將資料建立「深拷貝」**。因此任何自動的拷貝動作都可以被視為是對執行效能影響很小的。

所有權



變數與資料互動的方式：複製(Clone)

- 要是真的想深拷貝 String 在堆積上的資料而非僅是堆疊資料的話，可以使用一個常見的方法(method)叫做 **clone**。以下是 clone 方法運作的範例：

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

- 此程式碼能執行無誤，並明確作出了像上圖這樣的行為，也就是堆積資料的確被複製了一份。
- 當看到 clone 的呼叫，就會知道有一些特定的程式碼被執行且消費可能是相對昂貴的。可以很清楚地知道有些不同的行為正在發生。

所有權

只在堆疊上的資料：拷貝(Copy)

- 還有一個小細節還沒提到，也就是在使用整數時的程式碼。回想一下之前的範例是這樣寫的，它能執行而且是有效的：

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

- 但這段程式碼似乎和剛學的互相矛盾：有呼叫 clone，但 x 卻仍是有效的，沒有移動到 y。
- 原因是因為像整數這樣的型別**在編譯時是已知大小，所以只會存在在堆疊上**。所以要拷貝一份實際數值的話是很快的。這沒有任何理由要讓 x 在 y 建立後被無效化。換句話說，這邊沒有所謂淺拷貝與深拷貝的差別。所以這邊呼叫 clone 的話不會與平常的淺拷貝有啥不一樣，可以保持這樣就好。

所有權

只在堆疊上的資料：拷貝(Copy)

- **Rust有個特別的標記叫做Copy特徵(trait)可以用在標記像整數這樣存在堆疊上的型別(會在之後介紹特徵)。**如果一個型別有**Copy特徵**的話，一個變數在賦值給其他變數後仍然會是有效的。
- 如果一個型別有實作(implement)**Drop特徵**的話，Rust 不會允許讓此型別擁有 Copy 特徵。如果對某個型別在數值離開作用域時，需要再做特別處理的話，對此型別標註 Copy 特徵會在編譯時期產生錯誤。
- 所以哪些型別有實作 Copy 特徵呢？可以閱讀技術文件來知道哪些型別有，但基本原則是任何簡單地純量數值都可以實作 Copy，且不需要配置記憶體或任何形式資源的型別也有實作 Copy。以下是一些有實作 Copy 的型別：
  - 所有整數型別像是： u32。
  - 所有浮點數型別像是： f64。
  - 布林型別 bool，它只有數值 true 與 false。
  - 字元型別 char。
  - 元組，**不過包含的型別也都要有實作 Copy 才行**。比如 (i32, i32) 就有實作 Copy，但 (i32, String) 則無。

所有權

所有權與函式

- 傳遞數值給函式的方式和賦值給變數是類似的。**傳遞變數給函式會是移動或拷貝，就像賦值一樣。**

範例說明了變數如何進入且離開作用域：

```rust
fn main() {
    let s = String::from("hello");  // s 進入作用域

    takes_ownership(s);             // s 的值進入函式
                                    // 所以 s 也在此無效

    let x = 5;                      // x 進入作用域

    makes_copy(x);                  // x 本該移動進函式裡
                                    // 但 i32 有 Copy，所以 x 可繼續使用

} // x 在此離開作用域，接著是 s。但因為 s 的值已經被移動了
  // 它不會有任何動作

fn takes_ownership(some_string: String) { // some_string 進入作用域
    println!("{}", some_string);
} // some_string 在此離開作用域並呼叫 `drop`
  // 佔用的記憶體被釋放

fn makes_copy(some_integer: i32) { // some_integer 進入作用域
    println!("{}", some_integer);
} // some_integer 在此離開作用域，沒有任何動作發生
```

如果嘗試在呼叫 takes_ownership 後使用 s，Rust 會拋出編譯時期錯誤。這樣的靜態檢查可以免於犯錯。可以試試看在 main 裡哪裡可以使用 s 和 x，以及所有權規則如何防止寫錯。

所有權

回傳值與作用域

- 回傳值一樣能轉移所有權，底下範例和前一頁範例都加上了註解說明一個函式如何回傳些數值：

```
fn main() {
    let s1 = gives_ownership();         // gives_ownership 移動它的回傳值給 s1

    let s2 = String::from("哈囉");     // s2 進入作用域

    let s3 = takes_and_gives_back(s2);  // s2 移入 takes_and_gives_back
                                        // 該函式又將其回傳值移到 s3
} // s3 在此離開作用域並釋放
  // s2 已被移走，所以沒有任何動作發生
  // s1 離開作用域並釋放

fn gives_ownership() -> String {             // gives_ownership 會將他的回傳值
                                             // 移動給呼叫它的函式

    let some_string = String::from("你的字串"); // some_string 進入作用域

    some_string                              // 回傳 some_string 並移動給
                                             // 呼叫它的函式
}

// 此函式會取得一個 String 然後回傳它
fn takes_and_gives_back(a_string: String) -> String { // a_string 進入作用域

    a_string  // 回傳 a_string 並移動給呼叫的函式
}
```

變數的所有權每次都會遵從相同的模式：賦值給其他變數就會移動。當擁有堆積資料的變數離開作用域時，該數值就會被 drop 清除，除非該資料的所有權被移動到其他變數所擁有。

雖然這樣是正確的，但在每個函式取得所有權再回傳所有權的確有點囉唆。要是可以讓函式使用一個數值卻不取得所有權呢？要是想重複使用同個值，但每次都要傳入再傳出實在是很麻煩。而且時會想要讓函式回傳一些它們自己產生的值。

所有權

回傳值與作用域

- Rust 能讓用元組回傳多個數值，如底下範例所示：

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("'{}' 的長度為 {}。", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() 回傳 String 的長度

    (s, length)
}
```

- 但這實在太繁瑣，而且這樣的情況是很常見的。Rust 有提供一個概念能在不轉移所有權的狀況下使用數值，這叫做**參考(references)**。

所有權

```rust
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("'{}' 的長度為 {}。", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() 回傳 String 的長度

    (s, length)
}
```

參考與借用

- 在之前範例使用元組的問題在於，必須回傳String給呼叫的函式，才能繼續呼叫 calculate_length 之後繼續使用String，因為String會被傳入 calculate_length。不過其實可以提供個String數值的參考。
- **參考(references)就像是指向某個地址的指標**，可以追蹤存取到該處儲存的資訊，而該地址仍被其他變數所擁有。和指標不一樣的是，參考保證所指向的特定型別的數值一定是有效的。
- 以下是定義並使用 calculate_length 時，**在參數改用參考物件而非取得所有權**的程式碼：

```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("'{}' 的長度為 {}。", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

所有權

參考與借用

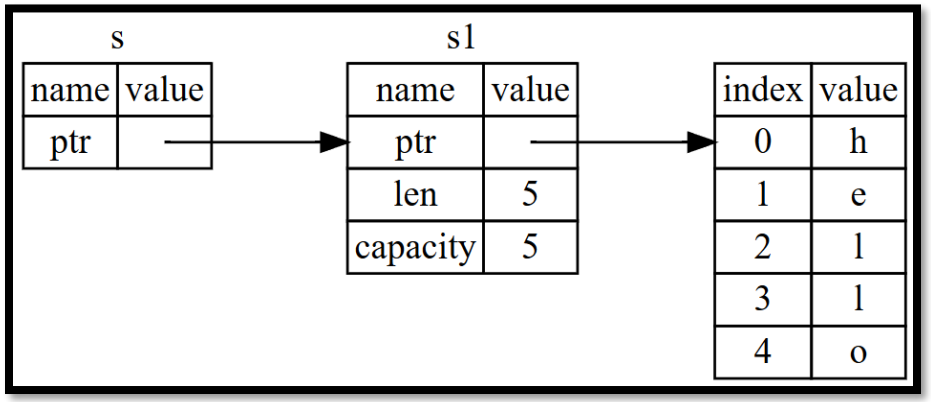
```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("'{}' 的長度為 {}。", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

- 首先會注意到原先變數宣告與函式回傳值會用到元組的地方都被更改了。再來注意到傳遞的是 **&s1 給 calculate_length**，然後在定義時是取 &String 而非 String。這些**「&」符號就是參考**，它們允許不必獲取所有權來參考它。以下用圖示示意：



顯示 &String s 指向 String s1 的示意圖

- **注意：使用 & 參考的反向動作是解參考(dereferencing)，使用的是解參考運算符號 *。會在之後看到一些解參考的範例並詳細解釋解參考。**

所有權

參考與借用

- 進一步看看函式的呼叫：

```
let s1 = String::from("hello");

let len = calculate_length(&s1);
```

- &s1 語法可以建立一個指向 s1 數值的參考，但不會擁有它。因為它並沒有所有權，它所指向的資料在不再使用參考後並不會被丟棄。同樣地，函式簽名也是用 & 說明參數 s 是個參考。加一些註解在範例上：

```
fn calculate_length(s: &String) -> usize { // s 是個 String 的參考
    s.len()
} // s 在此離開作用域，但因為它沒有它所指向的資料的所有權
  // 所以不會被釋放掉
```

- 變數 s 有效的作用域和任何函式參數的作用域一樣，但當不再使用參考時，參考所指向的數值不會被丟棄，因為沒有所有權。當函式使用參考作為參數而非實際數值時，不需要回傳數值來還所有權，因為不曾擁有過。
- 會稱呼建立參考這樣的動作叫做**借用(borrowing)**。就像現實世界一樣，如果有人擁有某項東西，可以借用給你。當使用完後，就還給他。你並不擁有它。

所有權

參考與借用

- 所以要是嘗試修改借用的東西會如何呢？請試試底下範例程式碼。**直接告訴你：它執行不了！**

```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
 --> src/main.rs:8:5
  |
7 | fn change(some_string: &String) {
  |                        ------- help: consider changing this to be a mutable referenc
8 |     some_string.push_str(", world");
| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `some_string` is a `&` reference, so the data it refer

For more information about this error, try `rustc --explain E0596`.
error: could not compile `ownership` due to previous error
```

- **如同變數預設是不可變，參考也是一樣的。不被允許修改參考的值。**

所有權

可變參考

- 可以修正以上範例的程式碼，使其可以變更借用的數值。加一點小修改，**改用可變參考就好**：

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

- 首先將 s 加上了 mut，然後在呼叫 change 函式的地方建立了一個可變參考 &mut s，然後更新函式的簽章成 some_string: &mut String 來接收這個可變參考。這樣能清楚表達 change 函式會改變它借用的參考。

所有權

可變參考

- **可變參考有個很大的限制：如果有一個數值的可變參考，就無法再對該數值有其他任何參考。**

  所以嘗試建立兩個 s 的可變參考的話就會失敗，如以下範例所示：

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

println!("{}, {}", r1, r2);
```

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src/main.rs:5:14
  |
4 |     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
5 |     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
6 |
7 |     println!("{}, {}", r1, r2);
  |                         -- first borrow later used here

For more information about this error, try `rustc --explain E0499`.
error: could not compile `ownership` due to previous error
```

- 此錯誤表示此程式碼是無效的，因爲無法同時可變借用 s 超過一次。第一次可變借用在 r1 且必須持續到它在 println! 用完爲止，但在其產生到使用之間，嘗試建立了另一個借用了與 r1 相同資料的可變借用 r2。

所有權

可變參考

- 這項防止同時間對相同資料進行多重可變參考的限制允許了可變行為，但是同時也受到一定程度的約束。這通常是新Rustaceans(程式開發者)遭受挫折的地方，因為多數語言都會隨意去改變其值。這項限制的好處是 Rust 可以在編譯時期就防止**資料競爭(data races)**。資料競爭和競爭條件(race condition)類似，它會由以下三種行為引發：

  1. 同時有兩個以上的指標存取同個資料。
  2. 至少有一個指標在寫入資料。
  3. 沒有針對資料的同步存取機制。

- 資料競爭會造成未定義行為(undefined behavior)，而且在執行時，通常是很難診斷並修正的。Rust 能夠阻止這樣的問題發生，不讓有資料競爭的程式碼編譯通過！

所有權

```rust
let mut s = String::from("hello");

{
    let r1 = &mut s;
} // r1 離開作用域，所以建立新的參考也不會有問題

let r2 = &mut s;
```

可變參考

- **可以用大括號來建立一個新的作用域來允許多個可變參考，只要不是同時擁有就好：**
- Rust 對於可變參考和不可變參考的組合中也實施著類似的規則，以下程式碼就會產生錯誤：

```rust
let mut s = String::from("hello");

let r1 = &s; // 沒問題
let r2 = &s; // 沒問題
let r3 = &mut s; // 很有問題！

println!("{}, {}, and {}", r1, r2, r3);
```

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:14
  |
4 |     let r1 = &s; // no problem
  |              -- immutable borrow occurs here
5 |     let r2 = &s; // no problem
6 |     let r3 = &mut s; // BIG PROBLEM
  |              ^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}, {}, and {}", r1, r2, r3);
  |                                -- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

- 看來也**不可以擁有不可變參考的同時擁有可變參考**。

所有權

可變參考

- 擁有不可變參考的使用者可不希望有人暗地裡突然改變了值！不過數個不可變參考是沒問題的，因為所有在讀取資料的人都無法影響其他人閱讀資料。**請注意參考的作用域始於它被宣告的地方，一直到它最後一次參考被使用為止**。舉例來說以下程式就可以編譯，**因為不可變參考最後一次的使用(println!)在可變參考宣告之前**：

```
let mut s = String::from("hello");

let r1 = &s; // 沒問題
let r2 = &s; // 沒問題
println!("{} and {}", r1, r2);
// 變數 r1 和 r2 將不再使用

let r3 = &mut s; // 沒問題
println!("{}", r3);
```

不可變參考 r1 和 r2 的作用域在 println! 之後結束。這是它們最後一次使用到的地方，也就是在宣告可變參考 r3 之前。它們的作用域沒有重疊，所以程式碼是允許的：**編譯器能辨別出參考何時在作用域之前不再被使用**。

雖然借用錯誤有時是令人沮喪的，但請記得這是Rust編譯器希望提前(在編譯時而非執行時)指出潛在程式錯誤並告訴問題的源頭在哪。這樣就不必親自追蹤為何資料跟預期的不一樣。

所有權

迷途參考

- 在有指標的語言中，通常都很容易不小心產生**迷途指標(dangling pointer)**。當資源已經被釋放但指標卻還留著，這樣的指標指向的地方很可能就已經被別人所有了。相反地，在Rust中編譯器會保證參考絕不會是迷途參考：如果有某些資料的參考，編譯器會確保資料不會在參考結束前離開作用域。來嘗試產生迷途指標，看看 Rust 怎麼產生編譯期錯誤：

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0106]: missing lifetime specifier
 --> src/main.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no value
help: consider using the `'static` lifetime
  |
5 | fn dangle() -> &'static String {
  |                 +++++++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `ownership` due to previous error
```

所有權

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0106]: missing lifetime specifier
 --> src/main.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is no value
help: consider using the `'static` lifetime
  |
5 | fn dangle() -> &'static String {
  |                 +++++++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `ownership` due to previous error
```

迷途參考

- 此錯誤訊息包含了一個還沒介紹的功能：生命週期(lifetimes)。會在之後討論生命週期。就算先不管生命週期的部分，錯誤訊息仍然告訴了程式出錯的關鍵點：

```
this function's return type contains a borrowed value, but there is no value
for it to be borrowed from
```

- 進一步看看 dangle 程式碼每一步發生了什麼：

```
fn dangle() -> &String { // 回傳 String 的迷途參考

    let s = String::from("hello"); // s 是個新 String

    &s // 我們回傳 String s 的參考
} // s 在此會離開作用域並釋放，它的記憶體就不見了。
  // 危險！
```

因為 s 是在 dangle 內產生的，當 dangle 程式碼結束時，s 會被釋放。但卻嘗試回傳參考。此參考會指向一個已經無效的 String。這看起來不太優！Rust 不允許這麼做

所有權

迷途參考

- 解決的辦法是直接回傳 String 就好：

```
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

- 這樣就沒問題了。所有權轉移了出去，沒有任何值被釋放。

參考規則

- 回顧討論到的參考規則：
  - 在任何時候，要嘛只能有一個可變參考，要嘛可以有任意數量的不可變參考。
  - 參考必須永遠有效。
- 接下來要來看看一個不太一樣的參考型別：切片(slices)。

所有權

切片型別

- 切片(slice)可以參考一串集合中的元素序列，而並非參考整個集合。**切片也算是某種類型的參考，所以它沒有所有權**。以下是個小小的程式問題：**寫一支函式接收一串用空格分開單字的字串，並回傳第一個找到的單字，如果函式沒有在字串找到空格的話，就代表整個字串就是一個單字，所以就回傳整個字串**。
- 先來想看看不使用切片的話，以下函式的簽名會長怎樣？這有助於理解切片想解決什麼問題：

```
fn first_word(s: &String) -> ?
```

所有權

切片型別

```
fn first_word(s: &String) -> ?
```

- 此函式 first_word 有一個參數 &String。不需要取得所有權，所以這是合理的。但回傳什麼呢？目前還沒有方法能夠描述一個字串的其中一部分。不過可以回傳單字的最後一個索引，也就是和空格作比較。像底下範例這樣試試看：

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

函式 first_word 回傳參數 String 第一個單字最後的索引

所有權

```rust
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

切片型別

- 因為需要走訪 String 的每個元素並檢查該值是否為空格，要用 as_bytes 方法將 String 轉換成一個位元組陣列：`let bytes = s.as_bytes();`
- 接下來用 iter 方法對位元組陣列建立一個**疊代器(iterator)**：`for (i, &item) in bytes.iter().enumerate() {`
- 會在後面討論疊代器的細節。現在只需要知道 iter 是個能夠回傳集合中每個元素的方法，然後 enumerate 會將 iter 的結果包裝起來回傳成元組(tuple)。**enumerate 回傳的元組中的第一個元素是索引，第二個才是元素的參考**。這樣比自己計算索引還來的方便。
- 既然 enumerate 回傳的是元組，可以用模式配對來解構元組。會在後面進一步解釋模式配對。所以在 for 迴圈中，指定了一個模式讓 i 取得索引然後 &item 取得元組中的位元組。因為從用 .iter().enumerate() 取得參考的，所以在模式中，用的是 & 來獲取。

所有權

切片型別

```rust
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

- 在 for 迴圈裡面使用字串字面值的語法搜尋位元組是不是空格。如果找到空格，就回傳該位置。不然就用 s.len() 回傳整個字串的長度。
- 現在有了一個能夠找到字串第一個單字結尾索引的辦法，但還有一個問題。回傳的是一個獨立的 usize，它套用在 &String 身上才有意義。換句話說，因為它是個與 String 沒有直接關係的數值，無法保證它在未來還是有效的。參考一下使用了之前範例中函式(first_word)的範例：

```rust
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // word 取得數值 5

    s.clear(); // 這會清空 String，這就等於 ""

    // word 仍然是數值 5 ，但是我們已經沒有相等意義的字串了
    // 擁有 5 的變數 word 現在完全沒意義！
}
```

先儲存呼叫函式 first_word的結果再變更 String 的內容

所有權

```rust
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // word 取得數值 5

    s.clear(); // 這會清空 String，這就等於 ""

    // word 仍然是數值 5 ，但是我們已經沒有相等意義的字串了
    // 擁有 5 的變數 word 現在完全沒意義！
}
```

切片型別

- 此程式可以成功編譯沒有任何錯誤，而且在呼叫 s.clear() 後仍然能使用 word。因為 word 和 s 並沒有直接的關係，word 在之後仍能繼續保留 5。可以用 s 取得 5 並嘗試取得第一個單字。但這樣就會是程式錯誤了，因為 s 的內容自從賦值 5 給 word 之後的內容已經被改變了。
- 要隨時留意 word 會不會與 s 的資料脫鉤是很煩瑣的且容易出錯！要是又寫了個函式 second_word，管理這些索引會變得非常難以管控！會不得不將函式簽名改成這樣：

```rust
fn second_word(s: &String) -> (usize, usize) {
```

- 現在得同時紀錄**起始**與**結束**的索引，而且還產生了更多與原本數值沒辦法直接相關的**計算結果**。**現在有三個非直接相關的變數需要保持同步**。Rust 為此提供了一個解決辦法：**字串切片(String slice)**。

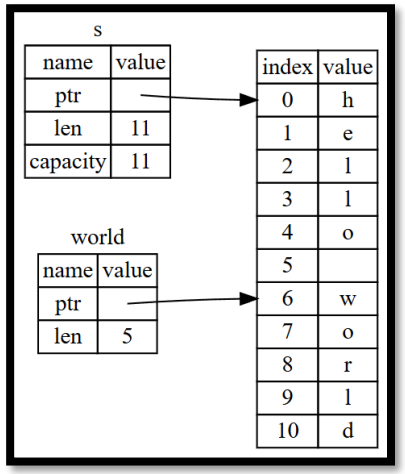
所有權

字串切片

```
let s = String::from("hello world");

let hello = &s[0..5];
let world = &s[6..11];
```

- 字串切片是 String 其中一部分的參考，它長得像這樣：
- 與其參考整個 String，hello 只參考了一部分的String，透過 [0..5] 來指示。可以像這樣 **[起始索引..結束索引]** 用中括號加上一個範圍來建立切片。**起始索引**是切片的第一個位置，而**結束索引**在索引結尾之後的位置(**所以不包含此值**)。在內部的切片資料結構會儲存起始位置，以及結束索引 與起始索引相減後的長度。所以用 let world = &s[6..11]; 作為例子的話， world 就會是個切片，包含一個指標指向索引爲 6 的位元組 s 和一個長度數值 5。



所有權

字串切片

- 要是想用 Rust 指定範圍的語法 .. 從索引 0 開始的話，可以省略兩個句點之前的值。換句話說，以下兩個是相等的：

```
let s = String::from("hello");

let slice = &s[0..2];
let slice = &s[..2];
```

- 同樣地，如果切片包含 String 的最後一個位元組的話，同樣能省略最後一個數值。這代表以下都是相等的：

```
let s = String::from("hello");

let len = s.len();

let slice = &s[3..len];
let slice = &s[3..];
```

所有權

字串切片

- 如果要獲取整個字串的切片，甚至能省略兩者的數值，以下都是相等的：

```
let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

- 注意：字串切片的索引範圍必須是有效的 UTF-8 字元界限。如果嘗試從一個多位元組字元(multibyte character)中產生字串切片，程式就會回傳錯誤。為了方便介紹字串切片，只使用了 ASCII 字元而已。 會在後面的「使用 String 儲存 UTF-8 編碼的文字」做更詳盡的討論。

所有權

字串切片

- 有了這些資訊，用切片來重寫 first_word 吧。對於「字串切片」的回傳型別會寫 &str：

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

- 如同之前範例一樣用判斷第一個空格取得了單字結尾的索引。當找到第一個空格，用字串的初始索引與當前空格的索引作為初始與結束索引來回傳字串切片。
- 現在當呼叫 first_word，就會取得一個與原本資料有直接相關的數值。此數值是由切片的起始位置即切片中的元素個數組成。

所有權

字串切片

- 這樣函式 second_word 一樣也可以回傳切片：

```
fn second_word(s: &String) -> &str {
```

- 現在有個不可能出錯且更直觀的 API，因為編譯器會確保 String 的參考會是有效的。**還記得在之前範例的錯誤嗎？就是那個當取得單字結尾索引，但字串卻已清空變成無效的錯誤。那段程式碼邏輯是錯誤的，卻不會馬上顯示錯誤。要是持續嘗試用該索引存取空字串的話，問題才會浮現**。切片可以讓這樣的程式錯誤無所遁形，並及早知道程式碼有問題。使用切片版本(first_word)的程式碼的話就會出現編譯期錯誤：

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // 錯誤！

    println!("第一個單字為：{}", word);
}
```

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
  --> src/main.rs:18:5
   |
16 |     let word = first_word(&s);
   |                           -- immutable borrow occurs here
17 |
18 |     s.clear(); // 錯誤！
   |     ^^^^^^^^^ mutable borrow occurs here
19 |
20 |     println!("第一個單字為：{}", word);
   |                               ---- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

所有權

字串切片

```rust
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // 錯誤！

    println!("第一個單字為：{}", word);
}
```

- 回憶一下借用規則，**要是有不可變參考的話，就不能取得可變參考。因為 clear 會縮減 String，它必須是可變參考**。在呼叫 clear 之後的 println! 用到了 word 的參考，所以不可變參考在該處仍必須保持有效。**Rust 不允許同時存在 clear 的可變參考與 word 的不可變參考**，所以編譯會失敗。Rust 不僅讓 API 更容易使用，還想辦法讓所有錯誤在編譯期就消除。

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
  --> src/main.rs:18:5
   |
16 |     let word = first_word(&s);
   |                           -- immutable borrow occurs here
17 |
18 |     s.clear(); // 錯誤！
   |     ^^^^^^^^^ mutable borrow occurs here
19 |
20 |     println!("第一個單字為：{}", word);
   |                                 ---- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

所有權

字串字面值作為切片

- 回想一下字串字面值是怎麼存在執行檔的。現在既然已經知道切片，就能知道更清楚理解字串字面值：

```
let s = "Hello, world!";
```

- 此處 s 的型別是 &str：它是指向執行檔某部份的切片。這也是為何字串字面值是不可變的，&str 是個不可變參考。

所有權

字串切片作為參數

- 知道可以取得字面值的切片與 String 數值後，可以再改善 first_word。也就是它的簽名表現：

```
fn first_word(s: &String) -> &str {
```

- 富有經驗的Rustacean(開發者)會用之前範例的方式編寫函式簽名，因為這讓該函式可以同時接受 &String 和 &str 的數值：

```
fn first_word(s: &str) -> &str {
```

- 如果有字串切片的話，可以直接傳遞。如果有 String 的話，可以傳遞此 String 的切片或參考。這樣的彈性用到了**強制解參考(deref coercion)**，這個功能會在之後的「函式與方法的隱式強制解參考」做介紹。

所有權

字串切片作為參數

- 定義函式的參數為字串切片而非 String 可以讓 API 更通用且不會失去任何功能：

```
fn main() {
    let my_string = String::from("hello world");

    // first_word 適用於 `String` 的切片，無論是部分或整體
    let word = first_word(&my_string[0..6]);
    let word = first_word(&my_string[..]);
    // first_word 也適用於 `String` 的參考，這等同於對整個 `String` 切片的操作。
    let word = first_word(&my_string);

    let my_string_literal = "hello world";

    // first_word 適用於字串字面值，無論是部分或整體
    let word = first_word(&my_string_literal[0..6]);
    let word = first_word(&my_string_literal[..]);

    // 因為字串字面值本來就是切片
    // 沒有切片語法也是可行的！
    let word = first_word(my_string_literal);
}
```

所有權

其他切片

- 字串切片如所想的一樣是特別針對字串的。但是還有更通用的切片型別。請考慮以下陣列：

```
let a = [1, 2, 3, 4, 5];
```

- 就像參考一部分的字串一樣，可以這樣參考一部分的陣列：

```
let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];

assert_eq!(slice, &[2, 3]);
```

- 此切片的型別為 &[i32]，它和字串運作的方式一樣，儲存了切片的第一個元素以及總長度。以後會對其他集合也使用這樣的切片。