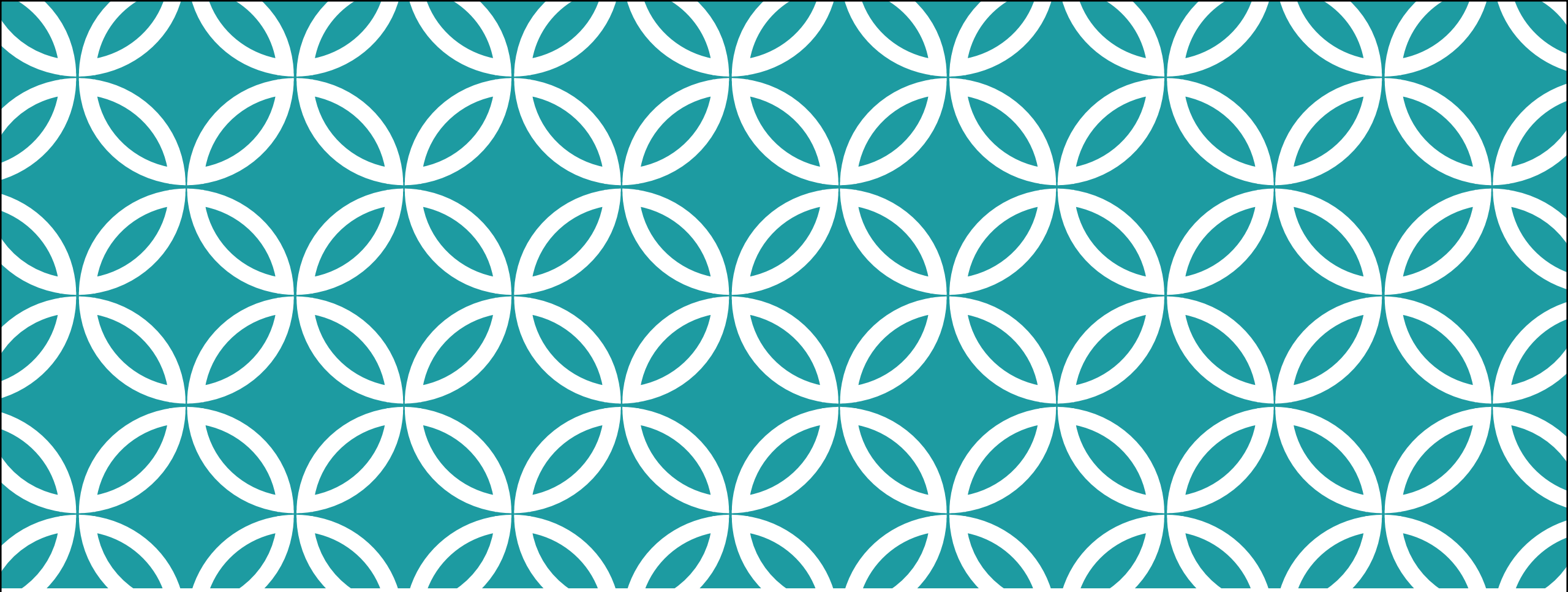

結構體(Struct)

結構體(Struct)

struct或結構體(structure)是個封裝並命名數個相關數值為單一組合的**自定型別**。如果熟悉物件導向語言的話，struct 就像是物件的資料屬性。在這邊會比較元組與結構體的差別，介紹如何使用結構體，並討論何時使用結構體組織資料是比較好的選擇。

將會解釋如何定義並產生結構體實例，也會討論如何**定義關聯函式，尤其是叫做方法(method)的關聯函式**，這能指定結構體型別特定的相關型別。結構體與將會在後面提到的列舉(enum)是 Rust 產生新型別的基本元件，它們能充分利用 Rust 的編譯時型別檢查。

結構體(Struct)

定義與實例化結構體

- 結構體(Structs)和元組類似，兩者都能持有多種相關數值。和元組一樣，結構體的每個部分可以是不同的型別。但與元組不同的地方是，**在結構體中必須為每個資料部分命名以便表達每個數值的意義**。因為有了這些名稱，結構體通常比元組還來的有彈性：不需要依賴資料的順序來指定或存取實例中的值。
- 欲定義結構體，輸入關鍵字 struct 並為整個結構體命名。結構體的名稱需要能夠描述其所組合出的資料意義。然後在大括號內，對每個資料部分定義名稱與型別，會稱為欄位(fields)。
- 舉例來說，底下範例定義了一個儲存使用者帳號的結構體：

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

結構體(Struct)

定義與實例化結構體

- 要在定義後使用該結構體，可以指定每個欄位的實際數值來建立結構體的實例(instance)。要建立實例的話，先寫出結構體的名稱再加上大括號，裡面會包含數個「key: value」的配對。**key 是每個欄位的名稱，而 value 就是想給予欄位的數值**。
- 欄位的順序可以不用和定義結構體時的順序一樣。換句話說，結構體的定義比較像是型別的通用樣板，然後實例會依據此樣板插入特定資料來將產生型別的數值。比如說，可以像底下範例這樣宣告一個特定使用者：

```
fn main() {
    let user1 = User {
        active: true,
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        sign_in_count: 1,
    };
}
```

結構體(Struct)

定義與實例化結構體

- 要取得結構體中特定數值的話，使用句點。如果想要此使用者的電子郵件信箱，使用 user1.email。如果該實例可變的話，可以使用句點並賦值給該欄位來改變其值。底下範例顯示了如何改變一個可變 User 實例中 email 欄位的值：

```
fn main() {
    let mut user1 = User {
        active: true,
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
}
```

- **請注意整個實例必須是可變的，Rust 不允許只標記特定欄位是可變的**。就像任何表達式一樣，可以在函式本體最後的表達式中，建立一個新的結構體實例作為回傳值。

結構體(Struct)

定義與實例化結構體

- 底下範例展示了 build_user 函式會依據給予的電子郵件和使用者名稱來回傳 User 實例。而 active 欄位取得數值 true 且 sign_in_count 取得數值 1：

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        email: email,
        username: username,
        sign_in_count: 1,
    }
}
```

- 函式參數名稱與結構體欄位名稱相同是非常合理的，但是要重複寫 email 和 username 的欄位名稱與變數就有點冗長了。如果結構體有更多欄位的話，重複寫這些名稱就顯得有些煩人了。幸運的是，的確有更方便的語法！

結構體(Struct)

用欄位初始化簡寫語法

- 由於之前範例的參數名稱與結構體欄位名稱相同，可以使用**欄位初始化簡寫(field init shorthand)**語法來重寫 build_user，讓它的結果相同但不必重複寫出 email 和 username，如底下範例所示：

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        email,
        username,
        sign_in_count: 1,
    }
}
```

- 在此建立了 User 結構體的實例，它有一個欄位叫做 email。希望用 build_user 函式中的參數 email 賦值給 email 欄位。然後因為 email 欄位與 email 參數有相同的名稱，只要寫 email 就好，不必寫 email: email。

結構體(Struct)

使用結構體更新語法從其他結構體建立實例

- 通常也會從其他的實例來產生新的實例，保留大部分欄位，不過修改一些欄位數值，這時可以使用**結構體更新語法(struct update syntax)**。
- 首先底下範例顯示了沒有使用更新語法來建立新的 User 實例 user2。設置了新的數值給 email，但其他欄位就使用之前範例建立的 user1 相同的值：

```
fn main() {
    // --省略--

    let user2 = User {
        active: user1.active,
        username: user1.username,
        email: String::from("another@example.com"),
        sign_in_count: user1.sign_in_count,
    };
}
```

結構體(Struct)

使用結構體更新語法從其他結構體建立實例

- 使用結構體更新語法，可以用較少的程式碼達到相同的效果，如右下範例所示： **.. 語法表示剩下沒指明的欄位都會取得與所提供的實例相同的值**。

```
fn main() {
    // --省略--

    let user2 = User {
        active: user1.active,
        username: user1.username,
        email: String::from("another@example.com"),
        sign_in_count: user1.sign_in_count,
    };
}
```

```
fn main() {
    // --省略--

    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
}
```

- 右上範例的程式碼產生的 user2 實例有不同 email，但是有與 user1 相同的 username、active 和 sign_in_count。..user1 加在最後面表示任何剩餘的欄位都會與 user1 對應欄位的數值相同，不過可以用任意順序指定多少想指定的欄位，不需要與結構體定義欄位的順序一樣。

結構體(Struct)

```
fn main() {
    // --省略--

    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
}
```

使用結構體更新語法從其他結構體建立實例

- 注意到結構體更新語法和賦值一樣使用 =，這是因為它也會轉移資料，就如同在「**變數與資料互動的方式：移動**」段落看到的一樣。在此範例中，**在建立 user2 之後就無法再使用 user1，因為 user1 的 username 欄位的 String 被移到 user2 了**。
- 如果同時給 user2 的 email 與 username 新的 String，這樣 user1 會用到的數值只會有 active 和 sign_in_count，這樣 user1 在 user2 就仍會有效。**因為 active 和 sign_in_count 都是有實作 Copy 特徵的型別，所以在「變數與資料互動的方式：clone」段落討論到的行為會造成影響**。

結構體(Struct)

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

使用無名稱欄位的元組結構體來建立不同型別

- Rust 還支援定義結構體讓它長得像是元組那樣，稱作元組結構體(tuple structs)。元組結構體仍然有定義整個結構的名稱，但是它們的欄位不會有名稱，它們只會有欄位型別而已。元組結構體的用途在於當想要為元組命名，好讓它跟其他不同型別的元組作出區別，以及對常規結構體每個欄位命名是冗長且不必要的時候。
- 要定義一個元組結構體，一樣先從 struct 關鍵字開始，其後再接著要定義的元組。舉例來說，右上是兩個使用元組結構體定義的 Color 和 Poin：
- 注意 black 與 origin 屬於不同型別，因為它們是不同的元組結構體實例。每個定義的結構體都是專屬於自己的型別，就算它們的欄位型別可能一模一樣。舉例來說，一個參數為 Color 的函式就無法接受 Point 引數，就算它們的型別都是三個 i32 的組合。除此之外，元組結構體實例和元組類似，可以將它們解構為獨立部分，也可以使用 **. 加上索引**來取得每個數值。

結構體(Struct)

無任何欄位的類單元結構體

- 也可以定義沒有任何欄位的結構體！這些叫做類單元結構體(unit-like structs)，因為它們的行為就很像在「元組型別」提論過的單元型別(unit type)() 類似。**類單元結構體很適合用在當要實作一個特徵(trait)或某種型別**，但沒有任何需要儲存在型別中的資料。會在之後介紹特徵。
- 以下的範例宣告並實例化一個類單元結構體叫做 AlwaysEqual：

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

- 使用 struct 關鍵字定義想要的名稱 AlwaysEqual，然後加上分號就好，不必再加任何括號！這樣就能一樣用 subject 變數取得一個 AlwaysEqual 的實例：直接使用定義的名稱，不用加任何括號。想像一下之後可以針對 AlwaysEqual 的實例實作與其他型別實例相同的行爲，像是爲了測試回傳已知的結果。不需要任何資料就能實作該行爲！能在之後看到如何定義特徵(trait)並對任何型別實作它們，這也包含類單元結構體。

結構體(Struct)

結構體資料的所有權

- 在之前範例的 User 結構體定義中，使用了擁有所有權的 String 型別，而不是 &str 字串切片型別。這邊是故意這樣選擇的，因為希望每個結構體的實例可以擁有它所有的資料，在整個結構體都有效時，資料也是有效的。
- 要在結構體中儲存別人擁有的資料參考是可行的，但這會用到生命週期(lifetimes)，會在之後才會談到。生命週期能確保資料參考在結構體存在期間都是有效的。要是沒有使用生命週期來用結構體儲存參考的話，會如以下出錯：

```
struct User {
    active: bool,
    username: &str,
    email: &str,
    sign_in_count: u64,
}
```

```
fn main() {
    let user1 = User {
        active: true,
        username: "someusername123",
        email: "someone@example.com",
        sign_in_count: 1,
    };
}
```

結構體(Struct)

結構體資料的所有權

```rust
struct User {
    active: bool,
    username: &str,
    email: &str,
    sign_in_count: u64,
}
```

```rust
fn main() {
    let user1 = User {
        active: true,
        username: "someusername123",
        email: "someone@example.com",
        sign_in_count: 1,
    };
}
```

- **編譯器會抱怨它需要生命週期標記**：

```
$ cargo run
   Compiling structs v0.1.0 (file:///projects/structs)
error[E0106]: missing lifetime specifier
 --> src/main.rs:3:15
  |
3 |     username: &str,
  |               ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct User<'a> {
2 |     active: bool,
3 ~     username: &'a str,
  |

error[E0106]: missing lifetime specifier
 --> src/main.rs:4:12
  |
4 |     email: &str,
  |            ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct User<'a> {
2 |     active: bool,
3 |     username: &str,
4 ~     email: &'a str,
  |

For more information about this error, try `rustc --explain E0106`.
error: could not compile `structs` due to 2 previous errors
```

在之後會討論如何修正這樣的錯誤，好讓可以在結構體中儲存參考。

但現在的話，先用有所有權的 String 而非 &str 參考來避免錯誤。

結構體(Struct)

使用結構體的程式範例

- 為了瞭解何時會想要使用結構體，來寫一支計算長方形面積的程式。會先從單一變數開始，再慢慢重構成使用結構體。用 Cargo 建立一個新的專案rectangles，它將接收長方形的長度與寬度，然後計算出長方形的面積。底下範例展示了在專案底下用其中一種方式寫出來的小程式：

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "長方形的面積為 {} 平方像素。",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target/debug/rectangles`
長方形的面積為 1500 平方像素。
```

雖然此程式碼成功呼叫 area 函式計算出長方形的面積，但可以做得更好，讓程式碼更簡潔且更易閱讀。

結構體(Struct)

使用結構體的程式範例

- 此程式碼的問題在 area 的函式簽名就能看出來：

```
fn area(width: u32, height: u32) -> u32 {
```

- area 函式有寬度與長度兩個參數，可用以計算長方形的面積。但在程式中，其參數相關性卻沒有表達出來。要是能將寬度與長度組合起來的話，會更容易閱讀與管理。可以使用之前提到的「元組型別」。

結構體(Struct)

使用元組重構

- 底下範例展示了程式用元組的另一種寫法：

```
fn main() {
    let rect1 = (30, 50);

    println!(
        "長方形的面積為 {} 平方像素。",
        area(rect1)
    );
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

- 一方面來說，此程式的確比較好。元組增加了一些結構，而現在只需要傳遞一個引數。但另一方面來說，此版本的閱讀性反而更差。元組無法命名它的元素，所以需要索引部分元組，讓計算變得比較不清晰。
- 在計算面積時，哪個值是寬度還是長度的確不重要。但如果要顯示出來的話，這就很重要了！會需要記住元組索引0是 width，然後元組索引1是 height。如果有其他人要維護這段程式碼的話，就也記住這件事才能使用程式碼。由於無法從程式碼表達出資料的意義，它就很容易產生錯誤。

結構體(Struct)

使用結構體重構：賦予更多意義

- 可以用結構體來為資料命名以賦予其意義。可以將元組轉換成一個有整體名稱且內部資料也都有名稱的結構體，如底下範例所示：

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "長方形的面積為 {} 平方像素。",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

結構體(Struct)

```rust
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "長方形的面積為 {} 平方像素。",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

使用結構體重構：賦予更多意義

- 在此定義了一個結構體叫做 Rectangle。在大括號內，定義了 width 與 height 的欄位，兩者型別皆為 u32。然後在 main 中，建立了一個寬度為 30 長度為 50 的 Rectangle 實例。
- 現在 area 函式使需要一個參數 rectangle，其型別為 Rectangle 結構體實例的不可變借用。如同之前提到的，希望借用結構體而非取走其所有權。這樣一來，main 能保留它的所有權並讓 rect1 繼續使用，這也是為何要在要呼叫函式的簽名中使用 &。
- area 函式能夠存取 Rectangle 中的 width 與 height 欄位(存取借用結構體實例的欄位不會移動欄位數值，這就是為何常看到結構體的借用)。area 函式簽名可以表達出想要做的事情了：使用 width 與 height 欄位來計算 Rectangle 的面積。這能表達出寬度與長度之間的關係，並且給了它們容易讀懂的名稱，而不是像元組那樣用索引 0 和 1。這樣清楚多了。

結構體(Struct)

使用推導(derive)特徵實現更多功能

- 現在要是能夠在除錯程式時能夠印出 Rectangle 的實例並看到它所有的欄位數值就更好了。

  底下範例嘗試使用之前提到的 println! 巨集，但是卻無法執行：

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {}", rect1);
}
```

- 當編譯此程式碼時，會得到錯誤訊息：

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
```

結構體(Struct)

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {}", rect1);
}
```

使用推導特徵實現更多功能

- println!巨集預設可以做各式各樣的格式化，大括號告訴println!要使用Display特徵的格式化方式：其輸出結果是用來給最終使用者使用的。目前遇過的基本型別預設都會實作Display，因為它們只有一種顯示方式(像是 1)能夠給使用者。但是對結構體來說 println! 要怎麼格式化輸出結果就會有點不明確了，因為顯示的方式就很有多種。**是要加上頓號嗎？是要印出大括號嗎？所有的欄位都要顯示出來嗎？**基於這些不確定因素，Rust 不會去猜要的是什麼，所以結構體預設並沒有 Display 的實作，也就無法使用 println! 與 {} 佔位符。
- 如果繼續閱讀錯誤訊息，會得到一些有幫助的資訊：

```
= help: the trait `std::fmt::Display` is not implemented for `Rectangle`
= note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
```

結構體(Struct)

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {}", rect1);
}
```

使用推導特徵實現更多功能

- 來試試看，println! 巨集的呼叫方式現在看起來應該會像這樣 println!("rect1 is {:?}", rect1);。在 println! 內加上 **:?** 這樣的標記指的是想要使用 Debug 特徵來作為輸出格式方式。
- Debug 特徵能印出對開發者有幫助的資訊，在除錯程式時可以看到它的數值。但是要是編譯這樣的程式的話，卻還是會得到錯誤：

```
error[E0277]: `Rectangle` doesn't implement `Debug`
```

- 不過同樣地，編譯器又給了有用的資訊：

```
= help: the trait `Debug` is not implemented for `Rectangle`
= note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for Rectangle`
```

結構體(Struct)

使用推導特徵實現更多功能

- Rust 的確有印出除錯資訊的功能，但是要針對結構體顯式實作出來才會有對應的功能。為此可以在結構體前加上屬性(attribute) #[derive(Debug)]，如底下範例所示：

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {:?}", rect1);
}
```

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle { width: 30, height: 50 }
```

- 現在當執行程式，不會再得到錯誤了，而且可以看到格式化後的輸出結果(右上)：

結構體(Struct)

使用推導特徵實現更多功能

- 雖然這不是非常好看的輸出格式，但是它的確顯示了實例中所有的欄位數值，這對除錯時會非常有用。不過如果結構體非常龐大的話，會希望輸出格式可以比較好閱讀。為此可以在 println! 的字串使用 {:#?} 而非 {:?}。在此例中使用 {:#?} 風格的話，輸出結果就會如下：

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle {
    width: 30,
    height: 50,
}
```

- 另一種使用 Debug 格式印出數值的方式是使用 **dbg! 巨集** 。這會拿走一個表達式的所有權(相較於 println! 只會拿參考)，印出該 dbg! 巨集在程式碼中呼叫的檔案與行數，以及該表達式的數值結果，最後回傳該數值的所有權。

結構體(Struct)

使用推導特徵實現更多功能

- 呼叫dbg!巨集會顯示到標準錯誤終端串流(stderr)，而不像println!是印到標準輸出終端串流(stdout)。會在之後的「將錯誤訊息寫入標準錯誤而非標準輸出」段落進一步討論 stderr 與 stdout。
- 以下的範例印出賦值給 width 的數值，以及整個 rect1 結構體的數值：

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

在表達式 30 * scale 加上 dbg!，因爲 dbg! 會回傳表達式的數值所有權，width 將能取得和不加上 dbg! 時相同的數值。而不希望 dbg! 取走 rect1 的所有權，所以在下一個 rect1 的呼叫使用參考。以下是此範例得到的輸出結果：

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.61s
     Running `target/debug/rectangles`
[src/main.rs:10] 30 * scale = 60
[src/main.rs:14] &rect1 = Rectangle {
    width: 60,
    height: 50,
}
```

結構體(Struct)

```rust
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.61s
     Running `target/debug/rectangles`
[src/main.rs:10] 30 * scale = 60
[src/main.rs:14] &rect1 = Rectangle {
    width: 60,
    height: 50,
}
```

使用推導特徵實現更多功能

- 可以看見第一個輸出結果來自第十行，也就是除錯表達式 30 * scale 的地方，其結果數值爲 60 (整數實作的 Debug 格式只會印出它們的數值)。而在第十四行所呼叫的 dbg! 則輸出 &rect1 的數值，也就是 Rectangle 的結構體。此輸出就會使用 Rectangle 實作的 Debug 漂亮格式。當需要嘗試理解程式碼怎麼運作時，dbg! 巨集可以變得相當實用！
- 除了 Debug 特徵之外，Rust 還提供了一些特徵能透過 derive 屬性來使用並爲自訂型別擴增實用的行爲。會在之後介紹如何實作這些特徵的自訂行爲，以及如何建立自己的特徵。
- 函式 area 只會計算長方形的面積。這樣的行為要是能夠緊貼著 Rectangle 結構體，因為這樣一來它就不會相容於其他型別。看看如何繼續重構程式碼，**接下來可以將函式 area 轉換為 Rectangle 型別的方法(method)**。

結構體(Struct)

方法語法

- 方法(Methods)和函式類似，用 fn 關鍵字並加上它們名稱來宣告，它們都有參數與回傳值，然後它們包含一些程式碼能夠在其他地方呼叫方法。和函式不同的是，方法是針對結構體定義的(或是列舉和特徵物件，會在之後分別介紹它們)，且它們第一個參數永遠是 self，這代表的是呼叫該方法的結構體實例。

結構體(Struct)

定義方法

- 把 Rectangle 作為參數的 area 函式轉換成定義在 Rectangle 內的 area 方法，如底下範例所示：

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "長方形的面積為 {} 平方像素。",
        rect1.area()
    );
}
```

要定義Rectangle中的方法，先為Rectangle加個 impl(implementation)區塊來開始。所有在此區塊的內容都跟 Rectangle 型別有關。再來將 area 移入 impl 的大括號中，並將簽名中的第一個參數(在此例中是唯一一個)與其本體中用到的地方改成 self。

在 main 中原先使用 rect1 作為引數呼叫的 area，可以改成使用方法語法(method syntax)來呼叫 Rectangle 的 area 方法。方法語法在實例後面呼叫，在其之後加上句點、方法名稱、括號然後任何所需的引數。

結構體(Struct)

定義方法

- 把 Rectangle 作為參數的 area 函式轉換成定義在 Rectangle 內的 area 方法，如底下範例所示：

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "長方形的面積為 {} 平方像素。",
        rect1.area()
    );
}
```

在 area 的簽名中，使用 &self 而非 rectangle: &Rectangle。**&self 是 self: &Self 的簡寫**。在一個 impl 區塊內，Self 型別是該 impl 區塊要實作型別的別名。方法必須有個叫做 self 的 Self 型別作為它們的第一個參數，所以 Rust 在寫第一個參數時能直接簡寫成 self。

注意到在 self 縮寫的前面仍使用 &，已表示此方法是借用 Self 的實例，就像在 rectangle: &Rectangle 做的一樣。就和其他參數一樣，方法可以選擇拿走 self 的所有權、像這裡借用不可變的 self 或是借用可變的 self。

結構體(Struct)

定義方法

- 把 Rectangle 作為參數的 area 函式轉換成定義在 Rectangle 內的 area 方法，如底下範例所示：

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "長方形的面積為 {} 平方像素。",
        rect1.area()
    );
}
```

之所以選擇 **&self** 的原因和之前函式版本的 **&Rectangle** 一樣，不想取得所有權，只想讀取結構體的資料，而非寫入它。如果想要透過方法改變實例的數值的話，會使用 **&mut self** 作為第一個參數。

而只使用 **self** 取得所有權的方法更是非常少見，這種使用技巧通常是為了想改變 **self** 成想要的樣子，並且希望能避免原本被改變的實例繼續被呼叫。

使用方法而非函式最大的原因是，除了可以使用方法語法而不必在方法簽名重複 **self** 的型別之外，其更具組織性。將所有一個型別所能做的事都放入 **impl** 區塊中了，而不必讓未來的使用者在茫茫函式庫中尋找 **Rectangle** 的功能。

結構體(Struct)

定義方法

- 另外還可以選擇將方法的名稱取作其結構體的其中一個欄位。舉例來說，也可以在 Rectangle 定義一個 width 方法：

```
impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    if rect1.width() {
        println!("長方形的寬度不為零，而是 {}", rect1.width);
    }
}
```

這裡選擇讓 width 方法判斷實例的 width 是否大於 0：如果是的話回傳 true；如果為 0 的話就回傳 false。

可以讓欄位與方法擁有相同的名稱，並作為任何用途使用。在 main 中，當在 rect1.width 後方加上**括號**，Rust 就會知道指的是 width 方法。當沒有使用括號時，Rust 會知道指的是 width 欄位。

結構體(Struct)

定義方法

- 另外還可以選擇將方法的名稱取作其結構體的其中一個欄位。舉例來說，也可以在 Rectangle 定義一個 width 方法：

```
impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    if rect1.width() {
        println!("長方形的寬度不為零，而是 {}", rect1.width);
    }
}
```

雖然不是必定的做法，但通常將方法名稱與欄位設為一樣時，希望它只回傳該欄位的數值而已。像這樣的方法稱為 getter，Rust 並不會像其他語言那樣自動為結構體欄位實作它們。Getter常用於將欄位隱藏起來，但提供一個公開方法並只限讀取該欄位，來做為該型別的公開API。會在之後討論什麼是公開與私有，以及如何設計方法或欄位為公開或私有的。

結構體(Struct)

擁有更多參數的方法

- 來練習再實作另一個 Rectangle 的方法。這次要 Rectangle 的實例可以接收另一個Rectangle實例，要是self 本身(第一個Rectangle)可以包含另一個Rectangle的話就回傳true，不然的話就回傳false。也就是希望定一個方法 **can_hold** ，如右邊範例所示：

```
rect1 能容納 rect2 嗎？true
rect1 能容納 rect3 嗎？false
```

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!("rect1 能容納 rect2 嗎？{}", rect1.can_hold(&rect2));
    println!("rect1 能容納 rect3 嗎？{}", rect1.can_hold(&rect3));
}
```

- 然後預期的輸出結果會如左上所示，因為 rect2 的兩個維度都比 rect1 小，但 rect3 比 rect1 寬：

結構體(Struct)

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

擁有更多參數的方法

- 知道要定義方法的話，它一定得在 impl Rectangle 區塊底下。方法的名稱會叫做 can_hold。它會取得另一個 Rectangle 的不可變參考作為參數。可以從程式碼呼叫方法的地方來知道參數的可能的型別：rect1.can_hold(&rect2) 傳遞了 &rect2，這是一個 rect2 的不可變參考，同時也是 Rectangle 的實例。
- 這是合理的，因為只需要讀取 rect2(而不是寫入，寫入代表需要可變參考)，且希望 main 能夠保持 rect2 的所有權，好讓之後能在繼續使用它來呼叫 can_hold 方法。can_hold 的回傳值會是布林值，然後實作細節會是檢查 self 的寬度與長度是否都大於其他 Rectangle 的寬度與長度。加入範例的 can_hold 方法到 impl 區塊中，如右上範例所示：
- 當用範例執行此程式碼的話，會得到預期的輸出結果。方法可以在參數 self 之後接收更多參數，而那些參數就和函式中的參數用法一樣。

結構體(Struct)

關聯函式

- 所有在 impl 區塊內的方法都屬於關聯函式(associated functions)，因為它們都與 impl 實作的型別相關。要是有方法不需要自己的型別實例的話，可以定義個沒有 self 作為它們第一個參數的關聯函式(因此不會被稱作方法)。**已經在 String 型別使用過 String::from 這種關聯函式了**。
- 不屬於方法的關聯函式很常用作建構子，來產生新的結構體實例。這通常會叫做 new，但是 new 其實不是特殊名稱，也沒有內建在語言內。舉例來說，可以提供一個只接收一個維度作為參數的關聯函式，讓它賦值給寬度與長度，使其可以用 Rectangle 來產生正方形，而不必提供兩次相同的值：

```
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
```

回傳型別中與函式本體中的 Self 關鍵字是 impl 關鍵字接著出現的型別別名，在此例中就是 Rectangle。

要呼叫關聯函式的話，使用 :: 語法並加上結構體的名稱。比方說 let sq = Rectangle::square(3);。此函式用結構體名稱作為命名空間，:: 語法可以用在關聯函式以及模組的命名空間，會在之後介紹模組。

結構體(Struct)

多重 impl 區塊

- 每個結構體都允許有數個 impl 區塊。舉例來說，之前的範例與底下的範例展示的程式碼是一樣的，它讓每個方法都有自己的 impl 區塊。

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

- 這邊確沒有將方法拆為 impl 區塊的理由，不過這樣的語法是合理的。會在之後介紹泛型型別與特徵，看到多重 impl 區塊是非常實用的案例。
- **結構體可以自訂對領域有意義的型別**。使用結構體可以讓每個資料部分與其他部分具有相關性，並為每個部分讓程式更好讀懂。**在impl區塊中，可以定義與型別有關的函式，而方法就是其中一種關聯函式，能指定結構體能有何種行為**。但是結構體並不是自訂型別的唯一方法： **Rust 列舉功能**讓工具箱可以再多一項可以使用的工具。