泛型型別、特徵與生命週期

泛型型別、特徵與生命週期

每個程式語言都有能夠高效處理**概念複製**的工具。在 Rust 此工具就是泛型(generics)：實際(值)型別或其他屬性的抽象替代。可以表達泛型的行為或是它們與其他泛型有何關聯，而不必在編譯與執行程式時知道它們實際上是什麼。

函式也可以接受一些泛型型別參數，而不是實際(值)型別像是 i32 或 String，就像函式有辦法能接收多種未知數值作為參數來執行相同程式碼。

事際上在之前的 Option<T>、 Vec<T> 和 HashMap<K, V> 以及 Result<T, E> 使用過泛型了。接下來會探索如何用泛型定義自己的型別、函式與方法！

泛型型別、特徵與生命週期

首先會先檢視如何提取參數來減少重複的程式碼。接著會以相同的技巧使用泛型將兩個只有**參數型別不同的函式轉變成泛型函式**。

還會解釋如何在結構體和列舉使用泛型型別。再來會學會如何**使用特徵(traits)來定義共同行為**。可以組合特徵與泛型型別來限制泛型型別只適用在有特定行為的型別，而不是任意型別。

最後會來介紹生命週期(lifetimes)：一種能讓編譯器知道參考如何互相關聯的泛型，生命週期能提供給編譯器更多關於借用數值的資訊，好讓它在更多情況下可以確保參考是有效的。

泛型型別、特徵與生命週期

提取函數來減少重複性

- 泛型可以用佔位符(placeholder)替代特定型別，來表示多重型別並減少程式碼的重複性。在深入泛型語法之前，先來看如何不用泛型型別的情況下，用提取函式的方式減少重複的程式碼。之後就會用相同的方式來提取泛型函式！和透過找出重複的程式碼來提取程式一樣也將找出重複的函式來轉成泛型。
- 先從底下範例中一支尋找列表中最大數字的程式開始：

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("最大數字為 {}", largest);
}
```

泛型型別、特徵與生命週期

提取函數來減少重複性

- 儲存整數列表到變數 number_list 並將列表第一個數字的參考放入變數 largest。接著走訪列表中的所有元素，如果目前數字比 largest 內儲存的數字還大的話，就會替代成該變數的參考。
- 不過如果目前數值小於或等於最大值的話，變數就不會被改變，程式會接續檢查列表中的下一個數字。在考慮完列表中的所有數字後，largest 就應該會指向最大數字，在此例就是 100。
- 現在要從**兩個不同的數字列表**中找到最大值，可以重複上面範例的程式碼，然後在程式中兩個不同的地方使用相同的邏輯，如右邊範例所示：

```rust
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("最大數字為 {}", largest);
}
```

```rust
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("最大數字為 {}", largest);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("最大數字為 {}", largest);
}
```

泛型型別、特徵與生命週期

提取函數來減少重複性

- 雖然這樣的程式碼能執行，寫出重複的程式碼很囉唆而且容易出錯。還得記住每次更新時就得一起更新各個地方。
- **要去除重複的部分，以建立一層抽象，定義一個可以處理任意整數列表作為參數的函式**。這樣的解決辦法讓程式更清晰，而且能抽象表達出從列表中尋找最大值這樣的概念。

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("最大數字為 {}", largest);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let mut largest = &number_list[0];

    for number in &number_list {
        if number > largest {
            largest = number;
        }
    }

    println!("最大數字為 {}", largest);
}
```

泛型型別、特徵與生命週期

提取函數來減少重複性

- 在底下範例中提取了尋找最大值的程式碼成一個函式叫做 **largest**。
- 然後呼叫函式來尋找之前範例兩個列表中最大的數字。還可以在未來對其他任何 i32 的列表使用此函式：

```
fn largest(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("最大數字為 {}", result);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&number_list);
    println!("最大數字為 {}", result);
}
```

泛型型別、特徵與生命週期

提取函數來減少重複性

- largest 函式有個參數 list 可以代表傳遞給函式的 i32 型別切片。所以當呼叫此函式時，程式可以依據傳入的特定數值執行。
- 總結來說，以下是將之前範例的程式碼轉換成右邊範例的步驟：
  1. 找出重複的程式碼。
  2. 將重複的程式碼提取置函式本體內，並指定函式簽名輸入與回傳數值。
  3. 更新重複使用程式碼的實例，改呼叫定義的函式。
- 接著將以相同的步驟使用泛型來減少重複的程式碼。就像函式本體可以抽象出 list 而不用特定數值，泛型允許程式碼執行抽象型別。
- 舉例來說，假設有兩個函式：一個會找出 i32 型別切片中的最大值而另一個會找出 char 型別切片的最大值。要如何刪除重複的部分呢？

```rust
fn largest(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("最大數字為 {}", result);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

    let result = largest(&number_list);
    println!("最大數字為 {}", result);
}
```

泛型型別、特徵與生命週期

泛型資料型別

- 使用泛型(generics)來建立項目的定義，像是函式簽名或結構體，之後可以使用在不同的實際資料型別。
- 先看看如何使用泛型定義函式、列舉與方法。然後會再來看泛型對程式碼的效能影響如何。
- 在函式中定義
  - 當要使用泛型定義函數時，通常會將泛型置於函式簽名中指定參數與回傳值資料型別的位置。這樣做能讓程式碼更具彈性並向呼叫者提供更多功能，同時還能防止重複程式碼。
  - 接續 largest 函式的例子，右邊範例展示了兩個都在切片上尋找最大值的函式。要使用泛型將它們融合成一個函式。

```
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("最大數字為 {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("最大字元為 {}", result);
}
```

泛型型別、特徵與生命週期

泛型資料型別

- 在函式中定義
  - **largest_i32** 函式和之前範例提取的函式一樣，都是尋找切片中最大的 i32。而 **largest_char** 函式則尋找切片中最大的 char。
  - 函式本體都擁有相同的程式碼，開始用泛型型別參數來消除重複的部分，轉變成只有一個函式吧。要在新定義的函式中參數化型別的話，需要為參數型別命名，就和在函式中的參數數值所做的一樣。可以用任何標識符來命名型別參數名稱。
  - 但習慣上會用 T，因為 Rust 的型別參數名稱都盡量很短，常常只會有一個字母，而且 Rust 對於型別命名的慣用規則是駝峰式大小寫(CamelCase)。所以 T 作為「type」的簡稱是大多數 Rust 程式設計師的選擇。

```rust
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("最大數字為 {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("最大字元為 {}", result);
}
```

泛型型別、特徵與生命週期

泛型資料型別

- 在函式中定義
  - 當函式本體使用參數時，必須在簽名中宣告參數名稱，編譯器才能知道該名稱代表什麼。同樣地，當要在函式簽名中使用型別參數名稱，必須在使用前宣告該型別參數名稱。要定義泛型 largest 函式的話，在函式名稱與參數列表之間加上尖括號，其內就是型別名稱的宣告，如以下所示：

```
fn largest<T>(list: &[T]) -> &T {
```

  - 可以這樣理解定義：**函式 largest 有泛型型別 T，此函式有一個參數叫做 list，它的型別為數值 T 的切片。largest 函式會回傳與型別 T 相同型別的參考數值。**

泛型型別、特徵與生命週期

泛型資料型別

- 在**函式**中定義
  - 底下範例顯示了使用泛型資料型別於函式簽名組合出的 largest 函式。此範例還展示了如何依序用 i32 和 char 的切片呼叫函式。**注意此程式碼尚未能編譯**，不過會在之後修改它：
  - 如果現在就編譯程式碼的話，會得到此錯誤：

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0369]: binary operation `>` cannot be applied to type `&T`
 --> src/main.rs:5:17
  |
5 |         if item > largest {
  |            ---- ^ ------- &T
  |            |
  |            &T
  |
help: consider restricting type parameter `T`
  |
1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
  |             ++++++++++++++++++++++

For more information about this error, try `rustc --explain E0369`.
error: could not compile `chapter10` due to previous error
```

```
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("最大數字為 {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("最大字元為 {}", result);
}
```

泛型型別、特徵與生命週期

泛型資料型別

- 在函式中定義

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0369]: binary operation `>` cannot be applied to type `&T`
 --> src/main.rs:5:17
  |
5 |         if item > largest {
  |            ---- ^ ------- &T
  |            |
  |            &T
  |
help: consider restricting type parameter `T`
  |
1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
  |             ++++++++++++++++++++++

For more information about this error, try `rustc --explain E0369`.
error: could not compile `chapter10` due to previous error
```

  - 提示文字中提到了 std::cmp::PartialOrd 這個特徵(trait)。會在下個段落來討論特徵。現在只需要知道 largest 本體**無法適用於所有可能的 T 型別**，因為想要在本體中比較型別 T 的數值，只能在**能夠排序的型別中**做比較。
  - 要能夠比較的話，標準函式庫有提供 std::cmp::PartialOrd 特徵可以針對型別來實作。照著提示文字的建議，限制 T 只對有實作 PartialOrd 的型別有效。這樣此範例就能編譯，因為標準函式庫有對 i32 與 char 實作 PartialOrd。

泛型型別、特徵與生命週期

泛型資料型別

- 在**結構體**中定義
  - 一樣能以 <> 語法來對結構體中一或多個欄位使用泛型型別參數。底下範例展示了定義 Point<T> 結構體並讓 x 與 y 可以是任意型別數值：

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

  - 在結構體定義使用泛型的語法與函式定義類似。首先，在結構體名稱後方加上尖括號，並在其內宣告型別參數名稱。接著能在原本指定實際資料型別的地方，使用泛型型別來定義結構體。

泛型型別、特徵與生命週期

```rust
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

泛型資料型別

- 在**結構體**中定義
  - 注意到使用了一個泛型型別來定義 Point<T>，此定義代表 Point<T> 是某型別 T 下之通用的，而且欄位 x 與 y 擁有相同型別，無論最終是何種型別。如果用不同的型別數值來建立 Point<T> 實例，程式碼會無法編譯，如底下範例所示：

```rust
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

泛型型別、特徵與生命週期

```rust
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

泛型資料型別

- 在**結構體**中定義
  - 在此例中，當賦值 5 給 x 時，讓編譯器知道 Point<T> 實例中的泛型型別 T 會是整數。然後將 4.0 賦值給 y，這應該要和 x 有相同型別，所以會獲得底下錯誤：

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0308]: mismatched types
 --> src/main.rs:7:38
  |
7 |     let wont_work = Point { x: 5, y: 4.0 };
  |                                      ^^^ expected integer, found floating-point number

For more information about this error, try `rustc --explain E0308`.
error: could not compile `chapter10` due to previous error
```

泛型型別、特徵與生命週期

泛型資料型別

- 在**結構體**中定義
  - 要將結構體 Point 的 x 與 y 定義成擁有不同型別卻仍然是泛型的話，可以使用**多個**泛型型別參數。
  - 舉例來說，在底下範例改變了 Point 的定義為擁有兩個泛型型別 T 與 U，x 擁有型別 T 而 y 擁有型別 U：

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

現在這些所有的 Point 實例都是允許的了！要在定義中使用多少泛型型別參數都沒問題，但用太多的話會讓程式碼難以閱讀。

如果發現程式碼需要使用大量泛型的話，這通常代表程式碼需要重新組織成更小的元件。

泛型型別、特徵與生命週期

泛型資料型別

- 在**列舉**中定義
  - 如同結構體一樣，可以定義列舉讓它們的變體擁有泛型資料型別。
  - 看看在標準函式庫提供的 Option<T> 列舉：

```
enum Option<T> {
    Some(T),
    None,
}
```

  - 此定義現在來說應該就說得通了。如同所看到的 Option<T> 列舉有個泛型型別參數 T 以及兩個變體：Some 擁有型別 T 的數值；而 None 則是不具任何數值的變體。使用 Option<T> 列舉可以表達出一個可能擁有的數值這樣的抽象概念。而且因為 Option<T> 是泛型，不管可能的數值型別為何，都能使用此抽象。

泛型型別、特徵與生命週期

泛型資料型別

- 在**列舉**中定義
  - 列舉也能有數個泛型型別。在之前所使用列舉 Result 的定義就是個例子：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

  - Result 列舉有兩個泛型型別 T 和 E 且有兩個變體：Ok 擁有型別 T 的數值；而 Err 擁有型別 E 的數值。這樣的定義很方便能表達 Result 列舉可能擁有一個成功的數值(回傳型別 T 的數值)或失敗的數值(回傳型別為 E 的錯誤值)。
  - 事實上這就是在之前範例開啟檔案的方式，當成功開啟檔案時的 T 就會是型別 std::fs::File，然後當開啟檔案會發生問題時 E 就會是型別 std::io::Error。
  - **當發現程式碼有許多結構體或列舉都只有儲存的值有所不同時，可以使用泛型型別來避免重複。**

泛型型別、特徵與生命週期

```rust
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

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

泛型資料型別

- 在**方法**中定義
  - 可以對結構體或列舉定義方法，並也可以使用泛型型別來定義。右上範例展示在之前範例定義的結構體 Point<T> 並實作了一個叫做 x 的方法：
  - 在這 Point<T> 定義了一個方法叫做 x 並回傳欄位 x 的資料參考。注意需要在 impl 宣告 T，才有 T 可以用來標明在替型別 Point<T> 實作其方法。
  - 在 impl 之後宣告泛型型別 T，Rust 可以識別出 Point 尖括號內的型別為泛型型別而非實際型別。其實可以選用不同的泛型參數名稱，而不用和結構體定義的泛型參數一樣，不過通常使用相同名稱還是比較常見。無論該泛型型別最終會是何種實際型別，任何方法在有宣告泛型型別的 impl 內，都會被定義成適用於各種型別實例。

泛型型別、特徵與生命週期

泛型資料型別

- 在**方法**中定義
  - 當在定義方法時，也可以**對泛型型別加上些限制**。舉例來說，可以**只針對 Point<f32> 的實例來實作方法，而非適用於任何泛型型別的 Point<T> 實例**。在底下範例使用了實例型別 f32 而沒有在 impl 宣告任何型別：

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

  - 此程式碼代表 Point<f32> 會有個方法 distance_from_origin，其他 Point<T> 只要 T 不是型別 f32 的實例都不會定義此方法。此方法測量點距離座標 (0.0, 0.0) 有多遠並使用只有浮點數型別能使用的數學運算。

泛型型別、特徵與生命週期

```rust
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

泛型資料型別

- 在**方法**中定義
  - 在結構體定義中的泛型型別參數不會總是和結構體方法簽名中的相同。
  - 舉例來說，右上範例在 Point 結構體中使用泛型型別 X1 和 Y1，但在 mixup 方法中就使用 X2 Y2 以便清楚辨別。該方法用self Point的x值(型別為 X1)與參數傳進來的Point的y值(型別為 Y2)來建立新的 Point 實例。
  - 在 main 中，定義了一個 Point，其 x 型別為 i32(數值為 5)，y 型別為 f64(數值為 10.4)。變數p2是個 Point 結構體，x 為字串切片(數值為 "Hello")，y 為 char(數值為 c)。
  - 在 p1 呼叫 mixup 並加上引數 p2 的話會給 p3，它的 x 會有型別 i32，因為 x 來自 p1。而且變數 p3 還會有型別為 char 的 y，因為 y 來自 p2。println! 巨集的呼叫就會顯示 p3.x = 5, p3.y = c。
  - 此例是為了展示一些泛型參數是透過 impl 宣告而有些則是透過方法定義來取得：
    - **泛型參數 X1 和 Y1 是宣告在 impl 之後，因為它們與結構體定義有關聯。**
    - **泛型參數 X2 和 Y2 則是宣告在 fn mixup 之後，因為它們只與方法定義有關聯。**

泛型型別、特徵與生命週期

泛型資料型別

```
let integer = Some(5);
let float = Some(5.0);
```

- 使用泛型的程式碼效能
  - 當 Rust 編譯此程式碼時中，會進行單型化。在此過程中，會讀取 Option<T> 實例中使用的數值並識別出兩種 Option<T>：一種是 i32 而另一種是 f64。接著它就會將 Option<T> 的泛型定義展開為兩種定義 i32 與 f64，以此替換函式定義為特定型別。
  - 單型化的版本看起來會像這樣(編譯器實際使用的名稱會和這邊示範的不同)：

```
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

泛型 Option<T> 會被替換成編譯器定義的特定定義。因為 Rust 會編譯泛型程式碼成個別實例的特定型別，使用泛型就不會造成任何執行時消耗。當程式執行時，它就會和親自寫重複定義的版本一樣。單型化的過程讓 Rust 的泛型在執行時十分有效率。

泛型型別、特徵與生命週期

特徵：定義共同行為

- 特徵(trait)會定義特定型別與其他型別共享的功能：
  1. 可以使用特徵定義來抽象出共同行為。
  2. 可以使用特徵界限(trait bounds)來指定泛型型別為擁有特定行為的任意型別。
- **注意：特徵類似於其他語言常稱作介面(interfaces)的功能，但還是有些差異。**
- 定義特徵
  - 一個型別的行為包含對該型別可以呼叫的方法。如果可以對不同型別呼叫相同的方法，這些型別就能定義共同行為了。**特徵定義是一個將方法簽名統整起來，來達成一些目的而定義一系列行為的方法。**
  - 舉例來說，如果有數個結構體各自擁有不同種類與不同數量的文字：
    - 結構體 NewsArticle 儲存特定地點的新聞故事；
    - 結構體 Tweet則有最多280字元的內容，且有個欄位來判斷是全新的推文、轉推或其他推文的回覆。

泛型型別、特徵與生命週期

特徵：定義共同行為

- 定義特徵
  - 想要建立一個多媒體聚集器函式庫 crate 叫 **aggregator** 來顯示可能存在 NewsArticle 或 Tweet 實例的資料總結。要達成此目的的話，需要每個型別的總結，且會呼叫該實例的 summarize 方法來索取總結。
  - 底下範例顯示了表達此行為的 Summary 特徵定義：(src/lib.rs)

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

泛型型別、特徵與生命週期

特徵：定義共同行為

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

- 定義特徵
  - 在此使用 trait 關鍵字定義一個特徵，其名稱為 Summary。也將特徵宣告成 pub 所以其他會依賴此函式庫的 crate 也能用到此特徵，之後會再看到其他範例。在大括號中，宣告方法簽名來描述有實作此特徵的型別行為，在此例就是：fn summarize(&self) -> String。
  - 在方法簽名之後，**並沒有加上大括號提供實作細節，而是使用分號。每個有實作此特徵的型別必須提供其自訂行為的方法本體**。編譯器會強制要求任何有 Summary 特徵的型別都要有定義相同簽名的 summarize 方法。
  - 特徵本體中可以有多個方法，每行會有一個方法簽名並都以分號做結尾。

泛型型別、特徵與生命週期

特徵：定義共同行為

- 為型別實作特徵
  - 現在已經用 Summary 特徵定義了所需的方法簽名。可以在多媒體聚集器的型別中實作它。
  - 右邊範例顯示了 NewsArticle 結構體實作 Summary 特徵的方式，其使用頭條、作者、位置來建立 summarize 的回傳值。
  - 至於結構體 Tweet，使用使用者名稱加上整個推文的文字來定義 summarize，因為推文的內容長度已經被限制在 280 個字元以內了。

```rust
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{} {} 著 ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

泛型型別、特徵與生命週期

特徵：定義共同行為

- 為型別實作特徵
  - 為一個型別實作一個特徵類似於實作一般的方法。不同的地方在於在 impl 之後加上的是想要實作的特徵，然後在用 for 關鍵字加上想要實作特徵的型別名稱。
  - 在 impl 的區塊內置入該特徵所定義的方法簽名，使用大括號並填入方法本體來為對特定型別實作出特徵方法的指定行為。

```
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{} {} 著 ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

泛型型別、特徵與生命週期

特徵：定義共同行為

- 為型別實作特徵
  - 現在，就能像呼叫正常方法一樣，來呼叫 NewsArticle 和 Tweet 實例的方法，如以下所示： 現在函式庫已經對 NewsArticle 和 Tweet 實作 Summary 特徵了，crate 的使用者能像平常呼叫方法那樣，對 NewsArticle 和 Tweet 的實例呼叫特徵方法。唯一的不同是使用者必須將特徵也加入作用域中。
  - 以下的範例展示執行檔 crate 如何使用 **aggregator** 函式庫 crate：

```rust
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{} {} 著 ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

```rust
use aggregator::{self, Summary, Tweet};

fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 則新推文：{}", tweet.summarize());
}
```

此程式碼會印出「1 則新推文：

horse_ebooks: of course, as you probably already know, people」。

泛型型別、特徵與生命週期

特徵：定義共同行為

- 預設實作
  - 有時候對特徵內的一些或所有方法定義預設行為是很實用的，而不必要求每個型別都實作所有方法。然後當對特定型別實作特徵時，可以保留或覆蓋每個方法的預設行為。
  - 在底下範例在 Summary 特徵內指定 summarize 方法的預設字串，而不必像之前範例只定義了方法簽名：

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(閱讀更多...)")
    }
}
```

  - 要使用預設實作來總結 NewsArticle 的話，可以指定一個空的 impl 區塊，像是 impl Summary for NewsArticle {}。

泛型型別、特徵與生命週期

特徵：定義共同行為

```rust
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(閱讀更多...)")
    }
}
```

- 預設實作
  - 沒有直接對 NewsArticle 定義 summarize 方法，因為使用的是預設實作並聲明對 NewsArticle 實作 Summary 特徵。所以最後仍然能在 NewsArticle 實例中呼叫 summarize，如以下所示：

```rust
let article = NewsArticle {
    headline: String::from("Penguins win the Stanley Cup Championship!"),
    location: String::from("Pittsburgh, PA, USA"),
    author: String::from("Iceburgh"),
    content: String::from(
        "The Pittsburgh Penguins once again are the best \
         hockey team in the NHL.",
    ),
};

println!("有新文章發佈！{}", article.summarize());
```

  - 此程式碼會印出 有新文章發佈！(閱讀更多...)。

泛型型別、特徵與生命週期

特徵：定義共同行為

- 預設實作
  - 建立預設實作不會影響之前範例中 Tweet 實作的 Summary。因為要取代預設實作的語法，與當沒有預設實作時實作特徵方法的語法是一樣的。
  - 預設實作也能呼叫同特徵中的其他方法，就算那些方法沒有預設實作。這樣一來，特徵就可以提供一堆實用的功能，並要求實作者只需處理一小部分就好。
  - 舉例來說：可以定義 Summary 特徵，使其擁有一個必須要實作的summarize_author 方法，以及另一個擁有預設實作會呼叫 summarize_author 的方法：

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(從 {} 閱讀更多...)", self.summarize_author())
    }
}
```

泛型型別、特徵與生命週期

特徵：定義共同行為

- 預設實作

```rust
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(從 {} 閱讀更多...)", self.summarize_author())
    }
}
```

  - 要使用這個版本的 Summary，只需要在對型別實作特徵時定義 summarize_author 就好：

```rust
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

```rust
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from(
        "of course, as you probably already know, people",
    ),
    reply: false,
    retweet: false,
};

println!("1 則新推文：{}", tweet.summarize());
```

  - 在定義 summarize_author 之後，可以在結構體 Tweet 的實例呼叫 summarize，然後 summarize 的預設實作會呼叫提供的 summarize_author。因為已經定義了summarize_author，且 Summary 特徵有提供 summarize 方法的預設實作，所以不必再寫任何程式碼。
  - 此程式碼會印出 1 則新推文：(從 @horse_ebooks 閱讀更多...)。
  - **注意要是對相同方法覆寫實作的話，就無法呼叫預設實作。**

泛型型別、特徵與生命週期

特徵：定義共同行為

- 特徵作為參數
  - 現在知道如何定義與實作特徵，可以來探討如何使用特徵來定義函式來接受多種不同的型別。會使用之前範例 中 NewsArticle 與 Tweet 實作的 Summary 特徵，來定義一個函式 notify 使用它自己的參數 item 來呼叫 summarize 方法，所以此參數的型別預期有實作 Summary 特徵。 為此可以使用 impl Trait 語法，如以下所示：

```
pub fn notify(item: &impl Summary) {
    println!("頭條新聞！{}", item.summarize());
}
```

  - 與其在 item 參數指定實際型別，用的是 impl 關鍵字並加上特徵名稱。這樣此參數就會接受任何有實作指定特徵的型別。在 notify 本體中就可以用 item 呼叫 Summary 特徵的任何方法，像是 summarize。
  - 可以呼叫 notify 並傳遞任何 NewsArticle 或 Tweet 的實例。但如果用其他型別像是 String 或 i32 來呼叫此程式碼的話會無法編譯，因為那些型別沒有實作 Summary。

泛型型別、特徵與生命週期

特徵：定義共同行為

- 特徵作為參數-**特徵界限語法**
  - impl Trait 語法看起來很直觀，不過它其實是一個更長格式的語法糖，這個格式稱之為「特徵界限(trait bound)」，它長得會像這樣：

```
pub fn notify<T: Summary>(item: &T) {
    println!("頭條新聞！{}", item.summarize());
}
```

  - 此格式等同於之前段落的範例，只是比較長一點。將特徵界限置於泛型型別參數的宣告中，在尖括號內接在冒號之後。
  - impl Trait 語法比較方便，且在簡單的案例中可以讓程式碼比較簡潔；而特徵界限語法則適合用於其他比較複雜的案例。舉例來說可以有兩個有實作 Summary 的參數，使用 impl Trait 語法看起來會像這樣：

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
```

泛型型別、特徵與生命週期

特徵：定義共同行為

- 特徵作為參數-**特徵界限語法**

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
```

  - 如果想要此函式允許item1和item2是不同型別的話，使用impl Trait的確是正確的(只要都有實作Summary)。

    不過如果希望兩個參數都是同一型別的話，就得使用特徵界限來表達，如以下所示：

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

  - 泛型型別 T 作為 item1 和 item2 的參數會限制函式，讓傳遞給 item1 和 item2 參數的數值型別必須相同。

泛型型別、特徵與生命週期

特徵：定義共同行為

- 特徵作為參數-**透過 + 來指定多個特徵界限**
  - 也可以指定不只一個特徵界限。假設還想要 notify 中的 item 不只能夠呼叫 summarize 方法，還能顯示格式化訊息的話，可以在 notify 定義中指定 item 必須**同時要有** Display 和 Summary。這可以使用 + 語法來達成：
    ```
    pub fn notify(item: &(impl Summary + Display)) {
    ```
  - + 也能用在泛型型別的特徵界限中：
    ```
    pub fn notify<T: Summary + Display>(item: &T) {
    ```
  - 有了這兩個特徵界限，notify 本體就能呼叫 summarize 以及使用 {} 來格式化 item。

泛型型別、特徵與生命週期

特徵：定義共同行為

- 特徵作為參數-**透過 where 來使特徵界限更清楚**
  - 使用太多特徵界限也會帶來壞處。每個泛型都有自己的特徵界限，所以有數個泛型型別的函式可以在函式名稱與參數列表之間包含大量的特徵界限資訊，讓函式簽名難以閱讀。
  - **因此 Rust 有提供另一個在函式簽名之後指定特徵界限的語法 where**。所以與其這樣寫：

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
```

  - 可以這樣寫 where 的語法，如以下所示：

```
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
```

  - 此函式簽名就沒有這麼複雜了，函式名稱、參數列表與回傳型別能靠得比較近，就像沒有一堆特徵界限的函式一樣。

泛型型別、特徵與生命週期

特徵：定義共同行為

- 回傳有實作特徵的型別
  - 也能在回傳的位置使用 impl Trait 語法來回傳某個有實作特徵的型別數值，如以下所示：

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    }
}
```

  - 將impl Summary作為回傳型別的同時，在函式returns_summarizable指定回傳有實作Summary特徵的型別而不必指出實際型別。在此例中，returns_summarizable回傳Tweet，但呼叫此函式的程式碼不需要知道。

泛型型別、特徵與生命週期

特徵：定義共同行為

- 回傳有實作特徵的型別
  - 回傳一個只有指定所需實作特徵的型別在閉包(closures)與疊代器(iterators)中非常有用，會在之後介紹它們。閉包與疊代器能建立只有編譯器知道的型別，或是太長而難以指定的型別。
  - **impl Trait 語法允許不用寫出很長的型別，而是只要指定函數會回傳有實作 Iterator 特徵的型別就好。**

```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from(
                "Penguins win the Stanley Cup Championship!",
            ),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                 hockey team in the NHL.",
            ),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from(
                "of course, as you probably already know, people",
            ),
            reply: false,
            retweet: false,
        }
    }
}
```

泛型型別、特徵與生命週期

特徵：定義共同行為

- 回傳有實作特徵的型別
  - 然而如果使用 impl Trait 的話，就只能回傳單一型別。舉例來說此程式碼指定回傳型別為 impl Summary ，但是寫說可能會回傳 NewsArticle 或 Tweet 的話就會無法執行。
  - 寫說可能回傳 NewsArticle 或 Tweet 的話是不被允許的，因為 **impl Trait 語法會限制在編譯器中最終決定的型別**。
  - 會在之後的「允許不同型別數值的特徵物件」來討論如何寫出這種行為的函式。

```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from(
                "Penguins win the Stanley Cup Championship!",
            ),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                 hockey team in the NHL.",
            ),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from(
                "of course, as you probably already know, people",
            ),
            reply: false,
            retweet: false,
        }
    }
}
```

泛型型別、特徵與生命週期

特徵：定義共同行為

- 透過特徵界限來選擇性實作方法
  - 在有使用泛型型別參數 impl 區塊中使用特徵界限，可選擇性地對有實作特定特徵的型別來實作方法。
  - 舉例來說：右邊範例的 Pair<T> 對所有 T 實作了 new 函式來回傳新的 Pair<T> 實例(Self 是 impl 區塊內的型別別名，在此就是 Pair<T>)。
  - 但在下一個 impl 區塊中，只有在其內部型別 T 有實作能夠做**比較的 PartialOrd 特徵**以及能夠**顯示在螢幕的 Display 特徵**的話，才會實作 cmp_display 方法：

```rust
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("最大的是 x = {}", self.x);
        } else {
            println!("最大的是 y = {}", self.y);
        }
    }
}
```

泛型型別、特徵與生命週期

特徵：定義共同行為

```
impl<T: Display> ToString for T {
    // --省略--
}
```

- 透過特徵界限來選擇性實作方法
  - 因為標準函式庫有此全面實作，可以在任何有實作 Display 特徵的型別呼叫ToString特徵的to_string方法。舉例來說，可以像這樣將整數轉變成對應的 String 數值，因為整數有實作 Display：

```
let s = 3.to_string();
```

  - **特徵與特徵界限能使用泛型型別參數來減少重複的程式碼的同時，告訴編譯器該泛型型別該擁有何種行為。編譯器可以利用特徵界限資訊來檢查程式碼提供的實際型別有沒有符合特定行為。**
  - 在動態語言中，要是呼叫一個該型別沒有的方法的話，會在執行時才發生錯誤。但是 Rust 將此錯誤移到編譯期間，必須在程式能夠執行之前確保有修正此問題。
  - 除此之外，還不用寫在執行時檢查此行為的程式碼，因為已經在編譯時就檢查了。這麼做可以在不失去泛型彈性的情況下，提升效能。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 生命週期(lifetime)是另一種已經使用過的泛型。不同於確保一個型別有沒有要的行為，**生命週期確保在需要參考的時候，它們都是有效的**。
- 在之前的「參考與借用」段落沒談到的是，Rust 中的每個參考都有個生命週期，這是決定該參考是否有效的作用域。**大多情況下生命週期是隱式且可推導出來的，就像大多情況下型別是可推導出來的。當多種型別都有可能時，就得詮釋型別**。
- 同樣地，當生命週期的參考能以不同方式關聯的話，就得詮釋生命週期。Rust 要求用泛型生命週期參數來詮釋參考之間的關係，以確保實際在執行時的參考絕對是有效的。
- 詮釋生命週期在大多數的程式語言中都沒有這個概念，所以這段可能會有點覺得陌生。雖然不會在此章涵蓋所有生命週期的內容，但是會講些可能遇到生命週期的常見場景，好更加熟悉這個概念。

泛型型別、特徵與生命週期

```
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

透過生命週期驗證參考

- 透過生命週期預防迷途參考
  - 生命週期最主要的目的就是要預防迷途參考(dangling references)，其會導致程式參考到其他資料，而非它原本想要的參考。請看一下右上範例的程式，它有一個外部作用域與內部作用域：
  - 注意：之前的範例在宣告變數時都沒有給予初始數值，所以變數名稱可以存在於外部作用域。乍看之下似乎違反 Rust 不存在空值的原則。但是如果嘗試在賦值前使用變數的話，就會獲得編譯期錯誤，這證明 Rust 的確不允許空值。

泛型型別、特徵與生命週期

```rust
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

透過生命週期驗證參考

- 透過生命週期預防迷途參考
  - 外部作用域宣告了一個沒有初始值的變數 r，然後內部作用域宣告了一個初始值為 5 的變數 x。在內部作用域中，嘗試將 x 的參考賦值給 r。然後內部作用域結束後，嘗試印出 r。
  - 此程式碼不會編譯成功，因為數值 r 指向的數值在嘗試使用它時已經離開作用域。以下是錯誤訊息。

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `x` does not live long enough
 --> src/main.rs:6:13
  |
6 |         r = &x;
  |             ^^ borrowed value does not live long enough
7 |     }
  |     - `x` dropped here while still borrowed
8 |
9 |     println!("r: {}", r);
  |                        - borrow later used here
```

變數 x 「存在的不夠久」。原因是因為當內部作用域在第 7 行結束時，x 會離開作用域。但是 r 卻還在外部作用域中有效，會說的「活得比較久」。如果 Rust 允許此程式碼可以執行的話，r 就會參考到 x 離開作用域後被釋放的記憶體位置，然後嘗試對 r 做的事情都不會是正確的了。所以 Rust 如何決定此程式碼無效呢？它使用了**借用檢查器**。

泛型型別、特徵與生命週期

```
fn main() {
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}                         // ---------+
```

透過生命週期驗證參考

- 借用檢查器
  - Rust 編譯器有個借用檢查器(borrow checker)會比較作用域來檢測所有的借用是否有效。底下範例顯示了之前範例的程式碼，但加上了變數生命週期的詮釋：
  - 在此定義 r 的生命週期詮釋為 'a 而 x 的生命週期為 'b。如同所見，內部的 'b 區塊比外部的 'a 生命週期區塊還小。
  - 在編譯期間，Rust 會比較兩個生命週期的大小，並看出 r 有生命週期 'a 但它參考的記憶體有生命週期 'b。程式被回絕的原因是因為 'b 比 'a 還短：被參考的對象比參考者存在的時間還短。

泛型型別、特徵與生命週期

```
fn main() {
    let r;                  // ---------+-- 'a
                            //          |
    {                       //          |
        let x = 5;          // -+-- 'b  |
        r = &x;             //  |       |
    }                       // -+       |
                            //          |
    println!("r: {}", r); //          |
}                           // ---------+
```

透過生命週期驗證參考

- 借用檢查器
  - 底下範例修正了此程式碼讓它不會存在迷途參考，並能夠正確編譯：

```
fn main() {
    let x = 5;            // ----------+-- 'b
                          //           |
    let r = &x;           // --+-- 'a  |
                          //   |       |
    println!("r: {}", r); //   |       |
                          // --+       |
}                         // ----------+
```

  - x 在此有生命週期 'b，此時它比 'a 還長。這代表 r 可以參考 x，因為 Rust 知道 r 的參考在 x 是有效的時候永遠是有效的。
  - 現在知道參考的生命週期，以及 Rust 如何分析生命週期以確保參考永遠有效了。現在來探索函式中參數與回傳值的泛型生命週期。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 函式中的泛型生命週期
  - 寫個回傳兩個字串切片中較長者的函式。此函式會接收兩個字串切片並回傳一個字串切片。在實作 longest 函式後，底下範例的程式碼應該要印出最長的字串為 abcd：

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("最長的字串為 {}", result);
}
```

  - 注意想要函式接收的是字串切片的參考，而不是字串，因為不希望 longest 函式會取得它參數的所有權。之前「字串切片作為參數」段落有提到為何範例的參數正是所想要使用的參數。

泛型型別、特徵與生命週期

```rust
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

透過生命週期驗證參考

- 函式中的泛型生命週期
  - 如果嘗試實作 longest 函式時，如右上範例所示，它不會編譯過：
  - 會看到以下關於生命週期的錯誤：

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does not say
  whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `chapter10` due to previous error
```

  - 提示文字表示回傳型別需要有一個泛型生命週期參數，因為Rust無法辨別出回傳的參考指的是 x 還是 y。事實上也不知道，因為函式本體中的 if 區塊會回傳 x 的參考而 else 區塊會回傳 y 的參考！

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 函式中的泛型生命週期
  - 當定義函式時，不知道傳遞進此函式的實際數值會是什麼，所以不知道到底是if或else的區塊會被執行。也不知道傳遞進來的參考實際的生命週期為何，所以無法像之前範例那樣觀察作用域，來判定回傳的參考會永遠有效。
  - 要修正此錯誤，要加上泛型生命週期參數來定義參考之間的關係，讓借用檢查器能夠進行分析。

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the signature does not say
  whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `chapter10` due to previous error
```

泛型型別、特徵與生命週期

透過生命週期驗證參考

```
&i32          // 一個參考
&'a i32       // 一個有顯式生命週期的參考
&'a mut i32 // 一個有顯式生命週期的可變參考
```

- 生命週期詮釋語法
  - 生命週期詮釋(Lifetime Annotation)不會改變參考能存活多久，它們僅描述了數個參考的生命週期之間互相的關係，而不會影響其生命週期。就像當函式簽名指定了一個泛型型別參數時，函式便能夠接受任意型別一樣。函式可以指定一個泛型生命週期參數，這樣函式就能接受任何生命週期。
  - 生命週期詮釋的語法有一點不一樣：生命週期參數的名稱必須以撇號 (') 作為開頭，通常全是小寫且很短，就像泛型型別一樣。大多數的人會使用名稱 'a 作為第一個生命週期詮釋。將生命週期參數置於參考的 & 之後，並使用空格區隔詮釋與參考的型別。
  - 右上是一些例子：沒有生命週期參數的 i32 參考、有生命週期'a 的 i32 參考以及有生命週期'a 的 i32 可變參考：
  - 只有自己一個生命週期本身沒有多少意義，因為該詮釋是為了告訴 Rust 數個參考的泛型生命週期參數之間互相的關係。來研究生命週期詮釋如何在 longest 函式中相互關聯吧。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 函式簽名中的生命週期詮釋
  - 要在函式簽名使用生命週期詮釋的話，需要在函式名稱與參數列表之間的尖括號內宣告泛型生命週期參數，就像泛型型別參數那樣。
  - 想在此簽名表達這樣的限制：**只要所有參數都要是有效的，那麼回傳的參考才也會是有效的。也就是參數的生命週期與回傳參考的生命週期是相關的**。會將生命週期命名為'a 然後將它加到每個參考，如底下範例所示：

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

  - 此程式碼能夠編譯成功並產生希望在之前範例的 main 函式中得到的結果。

泛型型別、特徵與生命週期

```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

透過生命週期驗證參考

- 函式簽名中的生命週期詮釋
  - 此函式簽名告訴 Rust 它有個生命週期 'a，函式的兩個參數都是字串切片，並且會有生命週期'a。此函式簽名還告訴了 Rust 從函式回傳的字串切片也會和生命週期 'a 存活的一樣久。
  - 實際上它代表 longest 函式回傳參考的生命週期與函式引數傳入時生命週期較短的參考的生命週期一樣。這樣的關係正是想讓 Rust 知道以便分析這段程式碼。
  - 注意當在此函式簽名指定生命週期參數時，不會變更任何傳入或傳出數值的生命週期。只是告訴借用檢查器應該要拒絕任何沒有服從這些約束的數值。
  - 注意到 longest 函式不需要知道x和y實際上會活多久，只需要知道有某個作用域會用'a取代來滿足此簽名。

泛型型別、特徵與生命週期

```rust
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

透過生命週期驗證參考

- 函式簽名中的生命週期詮釋
  - 當要在函式詮釋生命週期時，詮釋會位於函式簽名中，而不是函式本體。就像型別會寫在簽名中一樣，生命週期詮釋會成為函式的一部份。在函式簽名加上生命週期能讓 Rust 編譯器的分析工作變得更輕鬆。
  - 如果當函式的詮釋或呼叫的方式出問題時，編譯器錯誤就能限縮到程式碼中指出來。如果都改讓Rust編譯器去推導可能的生命週期關係的話，編譯器可能會指到程式碼真正出錯之後的好幾步之後。
  - 當向 longest 傳入實際參考時，'a 實際替代的生命週期為 x 作用域與 y 作用域重疊的部分。換句話說，**泛型生命週期 'a 取得的生命週期會等於 x 與 y 的生命週期中較短的**。
  - 因為將回傳的參考詮釋了相同的生命週期參數 'a，回傳參考的生命週期也會保證在 x 和 y 的生命週期較短的結束前有效。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 函式簽名中的生命週期詮釋
  - 來看看如何透過傳入不同實際生命週期的參考來使生命週期詮釋能約束 longest 函式，如底下範例所示：

```
fn main() {
    let string1 = String::from("很長的長字串");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("最長的字串為 {}", result);
    }
}
```

  - 在此例中 string1 在外部作用域結束前都有效，而 string2 在內部作用域結束前都有效，然後 result 會取得某個有效參考直到內部作用域結束為止。執行此程式的話，會看到借用檢查器認可此程式碼，它會編譯成功然後印出 **最長的字串為 很長的長字串**。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 函式簽名中的生命週期詮釋
  - 接下來寫一個範例能要求result生命週期的參考必須是兩個引數中**較短**的才行。移動變數 result 的宣告到外部作用域，但保留變數 result 的賦值與 string2 一樣在內部作用域。
  - 然後也將使用到 result 的 println! 移到外部作用域，緊接在內部作用域結束之後。如右上範例所示，此程式碼會編譯不過：
  - 當嘗試編譯此程式碼，會看到以下錯誤：

```
fn main() {
    let string1 = String::from("很長的長字串");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("最長的字串為 {}", result);
}
```

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `string2` does not live long enough
 --> src/main.rs:6:44
  |
6 |         result = longest(string1.as_str(), string2.as_str());
  |                                            ^^^^^^^^^^^^^^^^ borrowed value does not live long
  enough
7 |     }
  |     - `string2` dropped here while still borrowed
8 |     println!("最長的字串為 {}", result);
  |                                 ------ borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error
```

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 函式簽名中的生命週期詮釋

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0597]: `string2` does not live long enough
 --> src/main.rs:6:44
  |
6 |         result = longest(string1.as_str(), string2.as_str());
  |                                            ^^^^^^^^^^^^^^^^ borrowed value does not live long
  enough
7 |     }
  |     - `string2` dropped here while still borrowed
8 |     println!("最長的字串為 {}", result);
  |                                 ------ borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `chapter10` due to previous error
```

  - 錯誤訊息表示要讓 result 在 println! 陳述式有效的話，string2 必須在外部作用域結束前都是有效的。Rust 會知道是因為在函式的參數與回傳值使用相同的生命週期 'a 來詮釋。
  - 能看出此程式碼的 string1 字串長度的確比 string2 長，因此 result 會包含 string1 的參考。因為 string1 尚未離開作用域，所以 string1 的參考在 println! 陳述式中仍然是有效的才對。然而編譯器在此情形會無法看出參考是有效的。所以才告訴 Rust longest 函式回傳參考的生命週期等同於傳入參考中較短的生命週期。這樣一來借用檢查器就會否決之前範例的程式碼，因為它可能會有無效的參考。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 深入理解生命週期
  - 要指定生命週期參數的方式取決於函式的行為。
  - 舉例來說如果改變函式 longest 的實作為永遠只回傳第一個參數而不是最長的字串切片，就不需要在參數 y 指定生命週期。以下的程式碼就能編譯：

```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

  - 指定生命週期參數 'a 給參數 x 與回傳型別，但參數 y 則沒有，因為 y 的生命週期與 x 和回傳型別的生命週期之間沒有任何關係。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 深入理解生命週期

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0515]: cannot return reference to local variable `result`
  --> src/main.rs:11:5
   |
11 |     result.as_str()
   |     ^^^^^^^^^^^^^^^^ returns a reference to data owned by the current function

For more information about this error, try `rustc --explain E0515`.
error: could not compile `chapter10` due to previous error
```

  - 當函式回傳參考時，回傳型別的生命週期參數必須符合其中一個參數的生命週期參數。如果回傳參考沒有和任何參數有關聯的話，代表它參考的是函式本體中的數值。但這會是迷途參考，因為該數值會在函式結尾離開作用域。請看看以下嘗試在函式 longest 的實作做法，它並不會編譯成功：

```
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("超長的字串");
    result.as_str()
}
```

  - 在這邊雖然有對回傳型別指定生命週期參數 'a，但此實作還是會失敗，因為回傳值的生命週期與參數的生命週期完全無關。右上是獲得的錯誤訊息：

泛型型別、特徵與生命週期

```
$ cargo run
   Compiling chapter10 v0.1.0 (file:///projects/chapter10)
error[E0515]: cannot return reference to local variable `result`
  --> src/main.rs:11:5
   |
11 |     result.as_str()
   |     ^^^^^^^^^^^^^^^^ returns a reference to data owned by the current function

For more information about this error, try `rustc --explain E0515`.
error: could not compile `chapter10` due to previous error
```

透過生命週期驗證參考

- 深入理解生命週期
  - 問題在於 result 會離開作用域並在 longest 函式結尾被清除。卻嘗試從函式中回傳 result 的參考。無法指定生命週期參數來改變迷途參考，而且 Rust 不會允許將建立迷途參考。
  - 在此例中，最好的解決辦法是回傳有所有權的資料型別而非參考，並讓呼叫的函式自行決定如何清理數值。
  - 總結來說，生命週期語法是用來連接函式中不同參數與回傳值的生命週期。一旦連結起來，Rust 就可以獲得足夠的資訊來確保記憶體安全的運算並防止會產生迷途指標或違反記憶體安全的操作。

泛型型別、特徵與生命週期

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("無法找到 '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

透過生命週期驗證參考

- 結構體定義中的生命週期詮釋
  - 目前為止，定義過的結構體都持有型別的所有權。結構體其實也能持有參考，不過會需要在結構體定義中每個參考加上生命週期詮釋。右上範例有個持有字串切片的結構體 ImportantExcerpt：
  - 此結構體有個欄位 part 並擁有字串切片參考。如同泛型資料型別，在結構體名稱之後的尖括號內宣告泛型生命週期參數，所以就可以在結構體定義的本體中使用生命週期參數。此詮釋代表 ImportantExcerpt 的實例不能比它持有的欄位 part 活得還久。
  - main 函式在此產生一個結構體 ImportantExcerpt 的實例並持有一個參考，其為變數 novel 所擁有的 String 中的第一個句子的參考。novel 的資料是在 ImportantExcerpt 實例之前建立的。
  - 除此之外，novel 在 ImportantExcerpt 離開作用域之前不會離開作用域，所以 ImportantExcerpt 實例中的參考是有效的。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 生命週期省略
  - 已經學到了每個參考都有個生命週期，而且需要在有使用參考的函式與結構體中指定生命週期參數。
  - 然而在之前範例**有函式可以不詮釋生命週期並照樣編譯成功**，在左下範例再展示一次：

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

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

    &s[..]
}
```

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

  - 此函式可以不用生命週期詮釋仍照樣編譯過是有歷史因素的：在早期版本的 Rust(1.0 之前)，此程式碼是無法編譯的，因為每個參考都得有顯式生命週期。在當時的情況下，此函式簽名會長得像右上這樣：

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 生命週期省略

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

  - 在寫了大量的 Rust 程式碼後，Rust 團隊發現 Rust 開發者會在特定情況反覆輸入同樣的生命週期詮釋。這些情形都是可預期的，而且可以遵循一些明確的模式。開發者將這些模式加入編譯器的程式碼中，所以借用檢查器可以依據這些情況自行推導生命週期，而不必顯式詮釋。
  - 這樣的歷史值得提起的原因是因為很可能會有更多明確的模式被找出來並加到編譯器中，意味著未來對於生命週期詮釋的要求會更少。
  - 被寫進 Rust 參考分析的模式被稱作**生命週期省略規則(lifetime elision rules)**。這些不是程式設計師要遵守的規則，而是一系列編譯器能去考慮的情形。而如果程式碼符合這些情形時，就不必顯式寫出生命週期。
  - 省略規則無法提供完整的推導：如果 Rust 能明確套用規則，但在這之後還是有參考存在模棱兩可的生命週期，編譯器就無法猜出剩餘參考的生命週期。編譯器不會亂猜，它會回傳錯誤，說明需要指定生命週期詮釋。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 生命週期省略

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

  - 在函式或方法**參數上的生命週期稱為輸入生命週期(input lifetimes)**，而在**回傳值的生命週期則稱為輸出生命週期(output lifetimes)**。
  - 當參考沒有顯式詮釋生命週期時，編譯器會用三項規則來推導它們。**第一個規則適用於輸入生命週期，而第二與第三個規則適用於輸出生命週期**。如果編譯器處理完這三個規則，卻仍有參考無法推斷出生命週期時，編譯器就會停止並回傳錯誤。這些適用於 fn 定義的規則一樣適用於 impl 區塊。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 生命週期省略

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

  - 第一個規則是編譯器會給予每個參考參數一個生命週期參數。換句話說，一個函式只有一個參數的話，就只會有一個生命週期：fn foo<'a>(x: &'a i32)；一個函式有兩個參數的話，就會有分別兩個生命週期參數：fn foo<'a, 'b>(x: &'a i32, y: &'b i32)，以此類推。
  - 第二個規則是如果剛好只有一個輸入生命週期參數，該參數就會賦值給所有輸出生命週期參數：fn foo<'a>(x: &'a i32) -> &'a i32。
  - 第三個規則是如果有多個輸入生命週期參數，但其中一個是 **&self 或 &mut self**，由於這是方法，self 的生命週期會賦值給所有輸出生命週期參數。此規則讓方法更容易讀寫，因為不用寫更多符號出來。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 生命週期省略
  - 假裝是編譯器。檢查這些規則並找出底下範例中函式 first_word 簽名中參考的生命週期。簽名的參考一開始沒有任何生命週期：
    ```
    fn first_word(s: &str) -> &str {
    ```
  - 接著編譯器檢查第一個規則，指明每個參數都有自己的生命週期。如往常一樣指定 'a，所以簽名會變成：
    ```
    fn first_word<'a>(s: &'a str) -> &str {
    ```
  - 然後第二個規則也適用，因為這裡剛好就一個輸入生命週期而已。第二個規則指明只有一個輸入生命週期的話，就會賦值給所有其他輸出生命週期。所以簽名現在變成這樣：
    ```
    fn first_word<'a>(s: &'a str) -> &'a str {
    ```
  - 現在此函式所有的參考都有生命週期了，而且編譯器可以繼續分析，不必要求程式設計師在此詮釋函式簽名的生命週期。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 生命週期省略
  - 再看看一個例子，這次是之前範例一開始沒有任何生命週期參數的 longest 函式：

```
fn longest(x: &str, y: &str) -> &str {
```

  - 先檢查第一項規則：每個參數都有自己的生命週期。這次有兩個參數，所以有兩個生命週期：

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

  - 可以看出來第二個規則並不適用，因為有不止一個輸入生命週期。而第三**個也不適用，因為 longest 是函式而非方法，其參數不會有 self** 。走訪這三個規則下來，仍然無法推斷出回傳型別的生命週期。
  - 這就是為何嘗試編譯之前範例的程式碼會出錯的原因：編譯器走訪生命週期省略規則，但仍然無法推導出簽名中所有參考的生命週期。
  - 因為第三個規則僅適用於方法簽名，接下來就會看看這種情況時的生命週期，看看為何第三個規則可以不必常常在方法簽名詮釋生命週期。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 在方法定義中的生命週期詮釋
  - 當在有生命週期的結構體上實作方法時，其語法類似於之前在範例中泛型型別參數的語法。宣告並使用生命週期參數的地方會依據它們是否與結構體欄位或方法參數與回傳值相關。
  - 結構體欄位的生命週期永遠需要宣告在 impl 關鍵字後方以及結構體名稱後方，因為這些生命週期是結構體型別的一部分。
  - 在 impl 區塊中方法簽名的參考可能會與結構體欄位的參考生命週期綁定，或者它們可能是互相獨立的。除此之外，**生命週期省略規則常常可以省略方法簽名中的生命週期詮釋**。
  - 看看之前範例定義過的 ImportantExcerpt 來作為範例。

泛型型別、特徵與生命週期

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

透過生命週期驗證參考

- 在方法定義中的生命週期詮釋
  - 首先使用一個方法叫做 level 其參數只有 self 的參考而回傳值是 i32，這不是任何參考：
  - 必須在 impl 之後宣告生命週期參數，並在型別名稱後使用該生命週期。但是**不必在 self 的參考加上生命週期詮釋**，因為其適用於第一個省略規則。
  - 以下是第三個生命週期省略規則適用的地方：

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("請注意：{}", announcement);
        self.part
    }
}
```

  - 這裡有兩個輸入生命週期，所以 Rust 用第一個生命週期省略規則給予 &self 和 announcement 它們自己的生命週期。然後因為其中一個參數是 &self，回傳型別會取得 &self 的生命週期，如此一來所有的生命週期都推導出來了。

泛型型別、特徵與生命週期

透過生命週期驗證參考

- 靜態生命週期
  - 其中有個特殊的生命週期 'static 需要進一步討論，這是指該參考可以存活在整個程式期間。所有的字串字面值都有'static 生命週期，可以這樣詮釋：
    ```
    let s: &'static str = "我有靜態生命週期。";
    ```
  - 此字串的文字會直接儲存在程式的執行檔中，所以永遠有效。因此所有的字串字面值的生命週期都是 'static。
  - 有時可能會看到錯誤訊息建議使用 'static 生命週期。但在對參考指明 'static 生命週期前，最好想一下該參考的生命週期是否真的會存在於整個程式期間，以及是否真的該活得這麼久。
  - 大多數錯誤訊息會建議 'static 生命週期的情況都來自於嘗試建立迷途參考或可用的生命週期不符。這樣的情況下，應該是要實際嘗試解決問題，而不是指明 'static 生命週期。

泛型型別、特徵與生命週期

組合泛型型別參數、特徵界限與生命週期

- 用一個函式來總結泛型型別參數、特徵界限與生命週期的語法：

```
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("公告！{}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

這是之前範例會回傳兩個字串切片較長者的 longest 函式。

不過現在它有個額外的參數 ann，使用的是泛型型別 T，它可以是任何在 where 中所指定有實作 Display 特徵的型別。

此額外參數會在 {} 的地方印出來，這正是為何 Display 的特徵界限是必須的。

因為生命週期也是一種泛型，生命週期參數 'a 與泛型型別參數 T 都宣告在函式名稱後的尖括號內。