智慧指標

智慧指標

指標(pointer)是一個將變數儲存記憶體位址的通用概念。此位址參考或者說是「指向」一些其他資料。

Rust 中最常見的指標種類就是之前介紹的**參考(reference)**。參考以 **&** 符號作為指示並借用它們指向的數值。它們除了參考資料以外，沒有其他的特殊能力，也沒有任何額外開銷(負擔)。

另一方面，**智慧指標(Smart pointers)是個不只會有像是指標的行為，還會包含擁有的詮釋資料與能力**。智慧指標的概念並不是 Rust 獨有的，智慧指標起源於 C++ 且也都存在於其他語言。

Rust 在標準函式庫中有提供許多不同的智慧指標，不只能參考還具備更多的功能。為了探索各個概念，會來研究一些各種不同的智慧指標範例，包含參考計數(reference counting)智慧指標型別：**此指標允許一個資料可以有多個擁有者，並追蹤擁有者的數量，當沒有任何擁有者時，就清除資料。**

在 Rust 中，有所有權與借用的概念，所以參考與智慧指標之間還有一項差別：**參考是只有借用資料的指標，但智慧指標在很多時候都擁有它們指向的資料。**

智慧指標

雖然在前面沒有這樣稱呼，但已經在之前遇過一些智慧指標了，像是之前的 **String 和 Vec<T>**，雖然當時沒有稱呼它們為智慧指標。這些型別都算是智慧指標，因為它們都擁有一些記憶體並允許操控它們。它們也有詮釋資料以及額外的能力或保障：像是 String 就會將容量儲存在**詮釋資料**中，並**確保其資料永遠是有效的 UTF-8**。

智慧指標通常都使用結構體實作。和一般結構體不同，智慧指標會實作 **Deref 與 Drop** 特徵：

- **Deref 特徵允許智慧指標結構體的實例表現像是參考一樣，使其可以寫出能用在參考與智慧指標的程式碼。**
- **Drop 特徵允許自訂當智慧指標實例離開作用域時要執行的程式碼。**

在此們會討論這兩個特徵並解釋為何它們對智慧指標很重要。

智慧指標

有鑑於智慧指標在Rust是個常用的**通用設計模式**，在此不會涵蓋每一個現有的智慧指標。許多函式庫也都會提供它們自己的智慧指標，甚至能寫個自己的。會提及標準函式庫中最常用到的智慧指標：

- **Box<T>**：將數值配置到堆積上
- **Rc<T>**：參考計數型別來允許資料能有數個擁有者
- 透過**RefCell<T>**來存取**Ref<T>**與**RefMut<T>** ，這是在執行時而非編譯時強制執行借用規則的型別

除此之外，還會涵蓋到**內部可變性(interior mutability)**模式：讓不可變參考的型別能提供改變內部數值的API；還會討論**參考循環(reference cycles)**為何會導致記憶體泄漏以及如何預防它們。

智慧指標

使用 Box<T> 指向堆積上的資料

- 最直白的智慧指標是 box 其型別為 Box<T>。**Box 允許儲存資料到堆積上，而不是堆疊。留在堆疊上的會是指向堆積資料的指標**。可以回顧之前瞭解堆疊與堆積的差別。
- Box沒有額外的效能開銷，就只是將它們的資料儲存在堆積上而非堆疊而已。不過相對地它們也沒有多少額外功能。大概會在這些場合用到它們：
  1. 當有個型別無法在編譯時期確定大小，而又想在需要知道確切大小的情況下使用該型別的數值。
  2. 當有個龐大的資料，而想要轉移所有權並確保資料不會被拷貝。
  3. 當想要擁有某個值，但只在意該型別有實作特定的特徵，而不是何種特定型別。

智慧指標

使用 Box<T> 指向堆積上的資料

- 會在「透過 Box 建立遞迴型別」段落解說第一種情形。
- 而在第二種情形，轉移龐大的資料的所有權可能會很花費時間，因為在堆疊上的話會拷貝所有資料。要改善此情形，可以用 box 將龐大的資料儲存在堆積上。這樣就只有少量的指標資料在堆疊上被拷貝，而其參考的資料仍然保留在堆積上的同個位置。
- 第三種情況被稱之為特徵物件(trait object)，在之後會在「允許不同型別數值的特徵物件」段落來討論此議題。

智慧指標

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

使用 Box<T> 儲存資料到堆積上

- 在討論 Box<T> 在堆積儲存空間上的使用場合前，會先介紹語法以及如何對 Box<T> 內儲存的數值進行互動。右上範例顯示如何使用 box 在堆積上儲存一個 i32 數值：
- 定義了變數 b 其數值為 Box 配置在堆積上指向的數值 5。程式在此例會印出 b = 5，在此例中可以用在堆疊上相同的方式取得 box 的資料。
- 就像任何有所有權的數值一樣，當 box 離開作用域時會釋放記憶體，在此例就是當 b 抵達main結尾的時候。釋放記憶體作用於 box(儲存在堆疊上)以及其所指向的資料(儲存在堆積上)。
- 將單一數值放在堆積上的確沒什麼用處，所以不會對這種類型經常使用 box。
- 在大多數情況下將像 i32 這種單一數值預設儲存在堆疊的確比較適合。

智慧指標

透過 Box 建立遞迴型別

- **遞迴型別(recursive type)的數值可以用相同型別的其他數值作為自己的一部分**。遞迴型別對 Rust 來說會造成問題，**因為Rust需要在編譯期間時知道型別佔用的空間**。
- 由於這種巢狀數值理論上可以無限循環下去，Rust 無法知道一個遞迴型別的數值需要多大的空間。然而 box 則有已知大小，所以將 box 填入遞迴型別定義中，就可以有遞迴型別了。
- 來探索 **cons list 來作為遞迴型別的範例**。這是個在函式程式語言中常見的資料型別，很適合作為遞迴型別的範例。要定義的 cons list 型別除了遞迴的部分以外都很直白，因此這個例子的概念在往後遇到更複雜的遞迴型別時會很實用。

智慧指標

透過 Box 建立遞迴型別

- 更多關於 Cons List 的資訊
  - cons list是個起源於 Lisp 程式設計語言與其方言的資料結構，用巢狀配對組成，相當於 **Lisp 版的鏈結串列(linked list)**。這名字來自於 Lisp 中的 cons 函式(「**construct function**」的縮寫)，它會從兩個引數建構一個新的配對，而這通常包含一個數值與另一個配對。對其呼叫 cons 就能建構出擁有遞迴配對的 cons list。
  - 舉例來說，底下是個 cons list 的範例，包含了用括號包起來的 1、2、3 列表配對：

```
(1, (2, (3, Nil)))
```

智慧指標

透過 Box 建立遞迴型別

- 更多關於 Cons List 的資訊 `(1, (2, (3, Nil)))`
  - 每個 cons list 的項目都包含兩個元素：目前項目的數值與下一個項目。
  - 列表中的最後一個項目只會包含一個數值叫做 Nil，並不會再連接下一個項目。
  - cons list 透過遞迴呼叫 cons 函式來產生。表示遞迴終止條件的名稱為 Nil。
  - 注意這和之前提到的「null」或「nil」的概念不全然相同，這些代表的是無效或空缺的數值。
  - 在 Rust 中 cons lists 不是常見的資料結構。大多數當在 Rust 需要項目列表時，Vec<T> 會是比較好的選擇。而其他時候夠複雜的遞迴資料型別確實在各種特殊情形會很實用，不過先從 cons list 開始的話，可以專注探討 box 如何定義遞迴資料型別。

智慧指標

透過 Box 建立遞迴型別

```
enum List {
    Cons(i32, List),
    Nil,
}
```

- 更多關於 Cons List 的資訊
  - 右上範例包含了 cons list 的列舉定義。注意到此程式碼還不能編譯過，因為 **List 型別並沒有已知的大小**。
  - 注意：定義的 cons list 只有 i32 數值是為了範例考量。當然可以使用之前討論過的泛型來定義它，讓 cons list 定義的型別可以儲存任何型別數值。
  - 使用 List 型別來儲存 1, 2, 3 列表的話會如底下範例的程式碼所示：

```
use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

  - 第一個 Cons 值會得到 1 與另一個 List 數值。此 List 數值是另一個Cons數值且持有 2 與另一個 List 數值。此 List 數值是另一個 Cons 數值且擁有 3 與一個 List 數值，其就是最後的 Nil，這是傳遞列表結尾訊號的非遞迴變體。

智慧指標

透過 Box 建立遞迴型別

- 更多關於 Cons List 的資訊
  - 如果嘗試編譯範例的程式碼，會得到底下的錯誤：

```
use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
  |               ---- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `List` representable
  |
2 |     Cons(i32, Box<List>),
  |               ++++    +
```

  - 錯誤顯示此型別的「**大小為無限**」，原因是因為**定義的 List 有個變體是遞迴**：它直接存有另一個相同類型的數值。所以 **Rust 無法判別出它需要多少空間才能儲存一個 List 的數值**。進一步研究為何會得到這樣的錯誤，首先來看 Rust 如何決定要配置多少空間來儲存非遞迴型別。

智慧指標

透過 Box 建立遞迴型別

- 計算非遞迴型別的大小
  - 回想一下之前在討論列舉定義時，在左下範例定義的 Message 列舉：

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

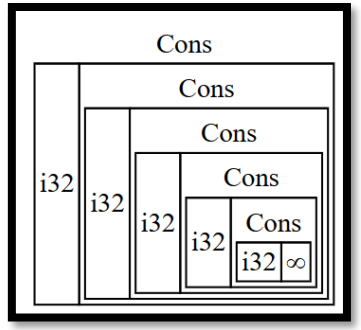
  - 要決定一個 Message 數值需要配置多少空間，Rust 會走訪每個變體來看哪個變體需要最大的空間。
  - Rust 會看到 Message::Quit 不佔任何空間、Message::Move 需要能夠儲存兩個 i32 的空間，以此類推。因為只有一個變體會被使用，一個 Message 數值所需的最大空間就是其最大變體的大小。

智慧指標

透過 Box 建立遞迴型別

```
enum List {
    Cons(i32, List),
    Nil,
}
```

- 計算非遞迴型別的大小
  - 將此對應到當 Rust 嘗試檢查像是之前範例的 List 列舉來決定遞迴型別需要多少空間時，會發生什麼事。編譯器先從查看 Cons 的變體開始，其存有一個 i32 型別與一個 List 型別。
  - 因此 Cons 需要的空間大小為 i32 的大小加上 List 的大小。
  - 為了要瞭解 List 型別需要的多少記憶體，編譯器再進一步看它的變體，也是從 Cons 變體開始。Cons 變體存有一個型別 i32 與一個型別 List，而這樣的過程就無限處理下去，如底下圖示所示：



智慧指標

透過 Box 建立遞迴型別

- 使用 Box<T> 取得已知大小的遞迴型別
  - 由於 Rust 無法判別出遞迴定義型別要配置多少空間，所以編譯器會針對此錯誤提供些實用的建議：

```
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `List` representable
  |
2 |     Cons(i32, Box<List>),
  |               ++++    +
```

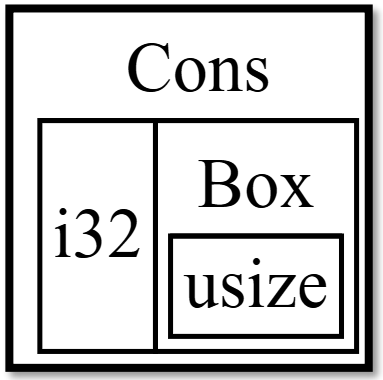
  - 在此建議中，**「indirection」代表與其直接儲存數值，可以變更資料結構，間接儲存指向數值的指標。**
  - 因為Box<T>是個指標，Rust永遠知道Box<T>需要多少空間：**指標的大小不會隨著指向的資料數量而改變。**這代表可以將 Box<T> 存入 Cons 變體而非直接儲存另一個 List 數值。
  - Box<T> 會指向另一個存在於堆積上的 List 數值而不是存在 Cons 變體中。概念上仍然有建立一個持有其他列表的列表，但此實作更像是將項目接著另一個項目排列，而非包含另一個在內。

智慧指標

透過 Box 建立遞迴型別

- 使用 Box<T> 取得已知大小的遞迴型別
  - 可以改變之前範例的 List 列舉定義以及 List 的使用方式，將其寫入底下範例，這次就能夠編譯過了：

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
}
```



  - Cons 變體需要的大小為 i32 加上儲存 box 指標的空間。Nil 變體沒有儲存任何數值，所以它需要的空間比 Cons 變體少。現在知道任何 List 數值會佔的空間都是一個 i32 加上 box 指標的大小。透過使用 box，打破了無限遞迴，所以編譯器可以知道儲存一個 List 數值所需要的大小。右上圖示顯示了 Cons 變體看起來的樣子。

智慧指標

透過 Box 建立遞迴型別

- 使用 Box<T> 取得已知大小的遞迴型別
  - Boxes只提供了**間接儲存與堆積配置**，**沒有其他任何特殊功能**，也沒有任何因這些特殊功能產生的額外效能開銷，所以它們很適合用於像是 cons list 這種只需要間接儲存的場合。
  - **Box<T> 型別是智慧指標是因為它有實作 Deref 特徵，讓 Box<T> 的數值可以被視為參考所使用。當 Box<T> 數值離開作用域時，該 box 指向的堆積資料也會被清除，因為其有 Drop 特徵實作。**
  - 這兩種特徵對於會討論的其他智慧指標型別所提供的功能，將會更加重要。來探討這兩種特徵的細節吧。

智慧指標

透過 Deref 特徵將智慧指標視為一般參考

- 實作Deref特徵可自訂**解參考運算子(dereference operator)***的行為(這不是相乘或全域運算子)。這種方式實作Deref的智慧指標可以被視為正常參考來對待，這樣操作參考的程式碼能用在智慧指標中。
  - 先看看解參考運算子如何在正常參考中使用。然後會嘗試定義一個行為類似 Box<T> 的自定型別，並看看為何解參考運算子無法像參考那樣用在新定義的型別。
  - 將會探討如何實作 Deref 特徵使智慧指標能像類似參考的方式運作。接著會看看 Rust 的強制解參考(deref coercion)功能並瞭解它如何處理參考與智慧指標。
  - 注意：即將定義的 MyBox<T> 型別與真正的 Box<T> 有一項很大的差別，就是不會將其資料儲存在堆積上。在此例會專注在 Deref 上，所以資料實際上儲存在何處，並沒有比指標相關行為來得重要。

智慧指標

```
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

追蹤指標的數值

- 一般的參考是一種指標，其中一種理解指標的方式是看成一個會指向存於某處數值的箭頭。在右上範例中建立了數值 i32 的參考，接著使用解參考運算子來追蹤參考的數值：
- 變數 x 存有 i32 數值 5。將 y 設置為 x 的參考。可以判定 x 等於 5。
- 不過要是想要判定 y數值的話，需要使用 *y 來追蹤參考指向的數值(也就是解參考)，這樣編譯器才能比較實際數值。
- 一旦解參考 y，就能取得 y 指向的整數數值並拿來與 5 做比較。

智慧指標

追蹤指標的數值

- 如果嘗試寫說 **assert_eq!(5, y);** 的話，會得到此編譯錯誤：

```
$ cargo run
   Compiling deref-example v0.1.0 (file:///projects/deref-example)
error[E0277]: can't compare `{integer}` with `&{integer}`
 --> src/main.rs:6:5
  |
6 |     assert_eq!(5, y);
  |     ^^^^^^^^^^^^^^^^ no implementation for `{integer} == &{integer}`
  |
  = help: the trait `PartialEq<&{integer}>` is not implemented for `{integer}`
  = note: this error originates in the macro `assert_eq` (in Nightly builds, run with -Z
  macro-backtrace for more info)
  = help: the following other types implement trait `PartialEq<Rhs>`:
            f32
            f64
            i128
            i16
            i32
            i64
            i8
            isize
          and 6 others
```

- 比較一個數字與一個數字的參考是不允許的，因為它們是不同的型別。
- 必須使用解參考運算子來追蹤其指向的數值。

智慧指標

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

像參考般使用 Box<T>

- 將之前範例的參考改用 Box<T> 重寫。右上範例對 Box<T> 使用解參考運算子的方式如就和之前範例對參考使用解參考運算子的方式一樣：
- 右上範例與之前範例主要的差別在於這裡設置 y 為一個指向 x 的拷貝數值的 Box<T> 實例，而不是指向 x 數值的參考。
- 在最後的判定中，可以對 Box<T> 的指標使用解參考運算子，跟對當 y 還是參考時所做的動作一樣。接下來，要來探討 Box<T> 有何特別之處，使其可以對自己定義的型別也可以使用解參考運算子。

智慧指標

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

定義自己的智慧指標

- 定義一個與標準函式庫所提供的 Box<T> 型別類似的智慧指標，並看看智慧指標預設行為與參考有何不同。然後就會來看能夠使用解參考運算子的方式。
- Box<T> 本質上就是定義成只有一個元素的元組結構體，所以右上範例用相同的方式來定義 MyBox<T>。也定義了 new 函式來對應於 Box<T> 的 new 函式。
- 定義了一個結構體叫做 MyBox 並宣告一個泛型參數 T，因為希望型別能存有任何型別的數值。MyBox 是個只有一個元素型別為 T 的元組結構體。
- MyBox::new 函式接受一個參數型別為 T 並回傳存有該數值的 MyBox 實例。

智慧指標

```
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

定義自己的智慧指標

- 將之前範例的 main 函式加到範例並改成使用定義的MyBox<T>型別而不是原本的Box<T>。

  右上範例的程式碼無法編譯，因為 Rust 不知道如何解參考MyBox：

- 以下是編譯結果出現的錯誤：

```
$ cargo run
   Compiling deref-example v0.1.0 (file:///projects/deref-example)
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
  --> src/main.rs:14:19
   |
14 |     assert_eq!(5, *y);
   |                   ^^
```

- MyBox<T> 型別無法解參考因為還沒有對型別實作該能力。
- 要透過 * 運算子來解參考的話，要實作 Deref 特徵。

智慧指標

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
```

透過實作 Deref 特徵來將一個型別能像參考般對待

- 要實作一個特徵的話，需要提供該特徵要求的方法實作。標準函式庫所提供的 Deref 特徵要求實作一個方法叫做deref，這會借用 self 並回傳內部資料的參考。
- 右上範例包含了對 MyBox 定義加上的 Deref 實作：
- type Target = T; 語法定義了一個供 Deref 特徵使用的關聯型別。
  - 關聯型別與宣告泛型參數會有一點差別，但是現在先不用擔心它們，會在之後深入探討。
- 對 deref 的方法本體加上 &self.0，deref 就可以回傳一個參考可以使用 * 運算子取得數值。.0 可以取的元組結構體的第一個數值。
- 之前範例的 main 函式現在對 MyBox<T> 數值的 * 呼叫就可以編譯了，而且判定也會通過！

智慧指標

```rust
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
```

透過實作 Deref 特徵來將一個型別能像參考般對待

- **沒有 Deref 特徵的話，編譯器只能解參考 & 的參考**。
- deref 方法讓編譯器能夠從任何有實作 Deref 的型別呼叫 deref 方法取得 & 參考，而它就可以進一步解參考獲取數值。
- 當在之前範例中輸入 *y 時，Rust 背後實際上是執行此程式碼：`*(y.deref())`
- Rust 將 * 運算子替換為方法 deref 的呼叫再進行普通的解參考，所以不必煩惱何時該或不該呼叫 deref 方法。
- 此 Rust 特性可以對無論是參考或是有實作 Deref 的型別都能寫出一致的程式碼。

智慧指標

```rust
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
```

透過實作 Deref 特徵來將一個型別能像參考般對待

- deref 方法會回傳一個數值參考，以及括號外要再加上普通解參考的原因，都是因為所有權系統。如果 deref 方法直接回傳數值而非參考數值的話，該數值就會移出 self。
- 不希望在此例或是大多數使用解參考運算子的場合下，取走 MyBox<T> 內部數值的所有權。
- 注意到每次在程式碼中使用*時，*運算子被替換成deref方法呼叫，然後再呼叫*剛好一次。
- 因為*運算子不會被無限遞迴替換，能剛好取得型別i32並符合範例assert_eq!中與 5 的判定。

智慧指標

函式與方法的隱式強制解參考

- 強制解參考(Deref coercion)會將有實作 Deref 特徵的型別參考轉換成其他型別的參考。舉例來說，強制解參考可以轉換 &String 成 &str，因為 String 有實作 Deref 特徵並能用它來回傳 &str。強制解參考是一個 Rust 針對函式或方法的引數的便利設計，且只會用在有實作 Deref 特徵的型別。
- 當將某個特定型別數值的參考作為引數傳入一個函式或方法，但該函式或方法所定義的參數卻不相符時，強制解參考就會自動發生，並進行一系列的 deref 方法呼叫，將提供的型別轉換成參數所需的型別。

智慧指標

函式與方法的隱式強制解參考

```rust
fn hello(name: &str) {
    println!("Hello, {name}!");
}
```

- Rust 會加入強制解參考的原因是因為程式設計師在寫函式與方法呼叫時，就不必加上許多**顯式參考 & 與解參考 ***。強制解參考還可以寫出能同時用於參考或智慧指標的程式碼。
- 為了展示強制解參考，使用之前範例定義的 MyBox<T> 型別以及所加上的 Deref 實作。右上範例中定義的函式使用字串切片作為參數：
- 可以使用字串切片作為引數來呼叫函式 hello，比方說 hello("Rust");。強制解參考可以透過 MyBox<String> 型別數值的參考來呼叫 hello，如底下範例所示：

```rust
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

智慧指標

函式與方法的隱式強制解參考

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

- 在此使用&m作為引數來呼叫函式 hello，這是 MyBox<String> 數值的參考。因為在之前範例有對 MyBox<T> 實作 Deref 特徵，Rust 可以呼叫 deref 將 &MyBox<String> 變成 &String。
- 標準函式庫對 String 也有實作 Deref 並會回傳字串切片，這可以在 Deref 的 API 技術文件中看到。所以 Rust 會再呼叫 deref 一次來將 &String 變成 &str，這樣就符合函式 hello 的定義了。
- 如果 Rust 沒有實作強制解參考的話，就得用底下範例的方式才能辦到之前範例使用型別 &MyBox<String> 的數值來呼叫 hello 的動作：

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&(*m)[..]);
}
```

智慧指標

函式與方法的隱式強制解參考

```
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&(*m)[..]);
}
```

- (*m) 會將 MyBox<String> 解參考成 String，然後 & 和 [..] 會從 String 中取得等於整個字串的字串切片，這就符合 hello 的簽名。沒有強制解參考的程式碼就難以閱讀、寫入或是理解，因為有太多的符號參雜其中。強制解參考能讓 Rust 自動做這些轉換。
- 當某型別有定義 Deref 特徵時，Rust 會分析該型別並重複使用 Deref::deref 直到能取得與參數型別相符的參考。Deref::deref 需要呼叫的次數會在編譯時期插入，所以使用強制解參考沒有任何的執行時開銷！

智慧指標

透過 Drop 特徵執行清除程式碼

- 第二個對智慧指標模式很重要的特徵是 **Drop**，**這能自訂數值離開作用域時的行為**。可以對任何型別實作Drop特徵，然後指定的程式碼就能用來釋放像是檔案或網路連線等資源。
- 在智慧指標介紹 Drop 的原因是因為 Drop 特徵的功能幾乎永遠會在實作智慧指標時用到。舉例來說，當 Box<T> 離開作用域時，它會釋放該 box 在堆積上指向的記憶體空間。
- 在某些語言中，當程式設計師使用完某些型別的實例後，每次都得呼叫釋放記憶體與資源程式碼。例子包括：檔案控制代碼(file handle)、socket或鎖。如果忘記的話，系統可能就會過載並崩潰。
- 在Rust 中可以對數值離開作用域時指定一些程式碼，然後編譯器就會自動插入此程式碼。就不用每次在特定型別實例使用完時，在程式每個地方都寫上清理程式碼。而且不會泄漏資源！

智慧指標

透過 Drop 特徵執行清除程式碼

- 透過實作 Drop 特徵可以指定當數值離開作用域時要執行的程式碼。Drop 特徵會要求實作一個方法叫做 drop，這會取得 self 的可變參考。為了觀察 Rust 何時會呼叫 drop，先用 println! 陳述式實作 drop。
- 右邊範例的結構體 CustomSmartPointer 只有一個功能，那就是在實例離開作用域時印出 Dropping CustomSmartPointer!。此範例能夠展示 Rust 何時會執行 drop 函式：

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("釋放 CustomSmartPointer 的資料 `{}`！", self.data);
    }
}

fn main() {
    let c = CustomSmartPointer {
        data: String::from("我的東東"),
    };
    let d = CustomSmartPointer {
        data: String::from("其他東東"),
    };
    println!("CustomSmartPointers 建立完畢。");
}
```

智慧指標

透過 Drop 特徵執行清除程式碼

- Drop 特徵包含在 prelude 中，所以不需要特地引入作用域。對CustomSmartPointer實作 Drop 特徵並提供會呼叫 println! 的 drop 方法實作。
- drop的函式本體用來放置想要在型別實例離開作用域時執行的邏輯。在此印出一些文字來展示 Rust 如何呼叫 drop。
- 在 main 中，建立了兩個 CustomSmartPointer 實例並印出 CustomSmartPointers 建立完畢。在 main 結尾，CustomSmartPointer實例會離開作用域，然後 Rust 就會呼叫放在 drop 方法的程式碼，也就是印出最終訊息。**注意到不需要顯式呼叫 drop 方法。**

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("釋放 CustomSmartPointer 的資料 `{}`！", self.data);
    }
}

fn main() {
    let c = CustomSmartPointer {
        data: String::from("我的東東"),
    };
    let d = CustomSmartPointer {
        data: String::from("其他東東"),
    };
    println!("CustomSmartPointers 建立完畢。");
}
```

智慧指標

透過 Drop 特徵執行清除程式碼

- 當執行此程式時，會看到以下輸出：

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.60s
     Running `target/debug/drop-example`
CustomSmartPointers 建立完畢。
釋放 CustomSmartPointer 的資料 `其他東東`!
釋放 CustomSmartPointer 的資料 `我的東東`!
```

- 當實例離開作用域時，Rust會自動呼叫drop，呼叫指定的程式碼。變數會以與建立時**相反**的順序被釋放，所以 d 會在 c 之前被釋放。
- 此範例給了一個觀察 drop 如何執行的視覺化指引，通常會指定該型別所需的清除程式碼，而不是印出訊息。

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("釋放 CustomSmartPointer 的資料 `{}`!", self.data);
    }
}

fn main() {
    let c = CustomSmartPointer {
        data: String::from("我的東東"),
    };
    let d = CustomSmartPointer {
        data: String::from("其他東東"),
    };
    println!("CustomSmartPointers 建立完畢。");
}
```

智慧指標

透過 std::mem::drop 提早釋放數值

- 不幸的是，**無法直接了當地取消自動 drop 的功能**。停用 drop 通常是不必要的，整個 Drop 的目的本來就是要能自動處理。**不過有些時候可能會想要提早清除數值**。
- 其中一個例子是使用智慧指標來管理**鎖**：可能會想要強制呼叫 drop 方法來釋放鎖，好讓作用域中的其他程式碼可以取得該鎖。Rust 不會手動呼叫 Drop 特徵的 drop 方法。
- 不過如果想要一個數值在離開作用域前就被釋放的話，可以使用標準函式庫提供的 std::mem::drop 函式來呼叫。

智慧指標

```rust
fn main() {
    let c = CustomSmartPointer {
        data: String::from("某些資料"),
    };
    println!("CustomSmartPointer 建立完畢。");
    c.drop();
    println!("CustomSmartPointer 在 main 結束前就被釋放了。");
}
```

透過 std::mem::drop 提早釋放數值

- 如果修改之前範例的 main 函式來手動呼叫 Drop 特徵的 drop 方法，如右上範例，會得到編譯錯誤：
- 當嘗試編譯此程式碼，會獲得以下錯誤：

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
error[E0040]: explicit use of destructor method
  --> src/main.rs:16:7
   |
16 |     c.drop();
   |     --^^^^--
   |     | |
   |     | explicit destructor calls not allowed
   |     help: consider using `drop` function: `drop(c)`
```

- 此錯誤訊息表示不允許顯式呼叫 drop。錯誤訊息使用了一個術語**解構子(destructor)**，這是通用程式設計術語中表達會清除實例的函式。解構子對應的術語就是建構子(constructor)，這會建立實例。Rust 中的 drop 函式就是一種特定的解構子。
- Rust 不讓顯式呼叫 drop，因為 Rust 還是會在 main 結束時自動呼叫 drop。這樣可能會導致**重複釋放(double free)的錯誤**，因為 Rust 可能會嘗試清除相同的數值兩次。

智慧指標

```
fn main() {
    let c = CustomSmartPointer {
        data: String::from("某些資料"),
    };
    println!("CustomSmartPointer 建立完畢。");
    drop(c);
    println!("CustomSmartPointer 在 main 結束前就被釋放了。");
}
```

透過 std::mem::drop 提早釋放數值

- **當數值離開作用域時，無法停用自動插入的 drop，而且無法顯式呼叫 drop 方法，所以如果必須強制讓一個數值提早清除的話，可以用 std::mem::drop 函式。**
- std::mem::drop 函式不同於 Drop 中的 drop 方法，傳入想要強制提早釋放的數值作為引數。此函式也包含在prelude，所以可以修改之前範例的main來呼叫drop函式，如右上範例所示：
- 執行此程式會印出以下結果：

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/drop-example`
CustomSmartPointer 建立完畢。
釋放 CustomSmartPointer 的資料 `某些資料`!
CustomSmartPointer 在 main 結束前就被釋放了。
```

智慧指標

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/drop-example`
CustomSmartPointer 建立完畢。
釋放 CustomSmartPointer 的資料 `某些資料`!
CustomSmartPointer 在 main 結束前就被釋放了。
```

透過 std::mem::drop 提早釋放數值

- 釋放 CustomSmartPointer 的資料`某些資料`! 這段文字會在 CustomSmartPointer 建立完畢。與 CustomSmartPointer 在 main 結束前就被釋放了。 文字之間印出，顯示 drop 方法會在那時釋放 c。
- 可以在許多地方使用 Drop 特徵實作所指定的程式碼，讓清除實例變得方便又安全。
  - 舉例來說：可以用它來建立自己的記憶體配置器！透過 Drop 特徵與 Rust 的所有權系統，不必去擔心要記得清理，因為 Rust 會自動處理。也不必擔心會意外清理仍在使用的數值：所有權系統會確保所有參考永遠有效，並確保當數值不再需要使用時只會呼叫 drop 一次。
- 現在看過 Box<T> 以及一些智慧指標的特性了，來看看一些其他定義在標準函式庫的智慧指標。

智慧指標

Rc<T> 參考計數智慧指標

- 在大多數的場合，所有權是很明確的：能確切知道哪些變數擁有哪些數值。然而還是有些情況會需要讓一個數值能有數個擁有者。
- 舉例來說，在圖解資料結構中，**數個邊可能就會指向同個節點，而該節點概念上就被所有指向它的邊所擁有**。節點直到沒有任何邊指向它，也就是沒有任何擁有者時才會被清除。
- 必須使用 Rust 的型別 Rc<T> 才能擁有**多重所有權**，這是參考計數(reference counting)的簡寫。Rc<T> 型別會追蹤參考其數值的數量來決定該數值是否還在使用中。如果數值沒有任何參考的話，該數值就可以被清除，因為不會產生任何無效參考。

智慧指標

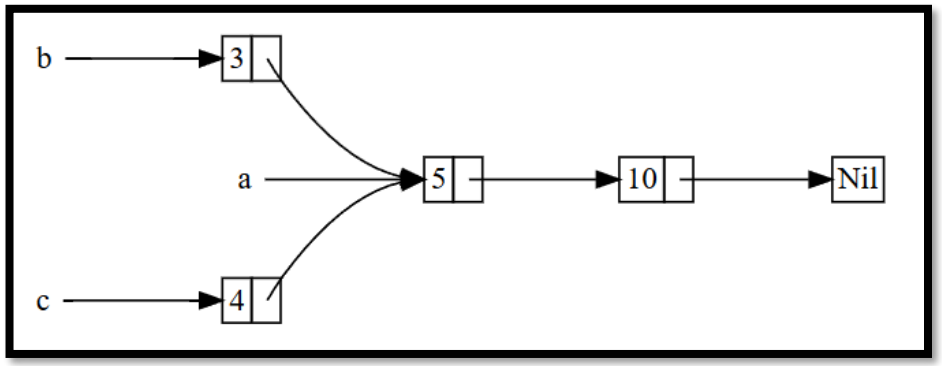
Rc<T> 參考計數智慧指標

- 想像 Rc<T> 是個在客廳裡的電視，當有人進入客廳要看電視時，它們就會打開它。其他人也能進來觀看電視。當最後一個人離開客廳時，它們會關掉電視，因為沒有任何人會再看了。如果當其他人還在看電視時，有人關掉了它，其他在看電視的人肯定會生氣。
- Rc<T> 型別的使用時機在於當想要在堆積上配置一些資料給程式中數個部分讀取，但是無法在編譯時期決定哪個部分會最後一個結束使用數值的部分。如果知道哪個部分會最後結束的話，可以將那個部分作為資料的擁有者就好，然後正常的所有權規則就會在編譯時生效。
- 注意到Rc<T>只適用於單一執行緒(single-threaded)的場合。之後討論並行(concurrency)時，會介紹如何在多執行緒程式達成參考計數。

智慧指標

使用 Rc<T> 來分享資料

- 回顧之前範例的 cons list 範例。回想一下當時適用 Box<T> 定義。這次會建立兩個列表，它們會同時共享第三個列表的所有權。概念上會如右下圖示所示：



- 會建立列表 a 來包含 5 然後是 10。然後會在建立兩個列表：b 以 3 為開頭而 c 以 4 為開頭。b 與 c 列表會同時連接包含 5 與 10 的第一個列表 a。換句話說，兩個列表會同時共享包含 5 與 10 的第一個列表。

智慧指標

使用 Rc<T> 來分享資料

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

- 嘗試使用 Box<T> 來定義這種情境的 List 的話會無法成功，如右上範例所示：
- 當編譯此程式碼，會得到以下錯誤：

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
error[E0382]: use of moved value: `a`
  --> src/main.rs:11:30
   |
9  |     let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
   |         - move occurs because `a` has type `List`, which does not implement the `Copy` trait
10 |     let b = Cons(3, Box::new(a));
   |                              - value moved here
11 |     let c = Cons(4, Box::new(a));
   |                              ^ value used here after move
```

- Cons 變體擁有它們存有的資料，所以當建立列表 b 時，a 會移動到 b，所以 b 就擁有 a。然後當嘗試再次使用 a 來建立 c 時，這就不會被允許，因為 a 已經被移走了。

智慧指標

使用 Rc<T> 來分享資料

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

- 可以嘗試改用參考來變更 Cons 的定義，但是這樣就必須指定生命週期參數。透過指定生命週期參數，會指定列表中的每個元素會至少活得跟整個列表一樣久。之前範例的元素和列表雖然可以這樣，但不是所有的場合都是如此。
- 最後可以改用 Rc<T> 來變更 List 的定義，如右上範例所示。每個 Cons 變體都會存有一個數值以及一個由 Rc<T> 指向的 List。
- 當建立 b 時，不會取走 a 的所有權，會clone a 存有的 Rc<List>，因而增加參考的數量從一增加到二，並讓 a 與 b 共享 Rc<List> 資料的所有權。也在建立 c 時clone a，增加參考的數量從二增加到三。每次呼叫 Rc::clone 時，對 Rc<List> 資料的參考計數就會成增加，然後資料不會被清除直到沒有任何參考為止。

智慧指標

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

使用 Rc<T> 來分享資料

- 需要使用 use 陳述式來將 Rc<T> 引入作用域，因為它沒有被包含在 prelude 中。
- 在 main 中，建立了一個包含 5 與 10 的列表並存入 a 的 Rc<List>。然後當建立 b 與 c 時，會呼叫函式 Rc::clone 來將 a 的 Rc<List> 參考作為引數傳入。
- 當然可以呼叫 a.clone() 而非 Rc::clone(&a)，但是在此情形中 Rust 的慣例是使用 Rc::clone。Rc::clone 的實作不會像大多數型別的 clone 實作會深度拷貝(deep copy)所有的資料。**Rc::clone 的呼叫只會增加參考計數，這花費的時間就相對很少。深拷貝通常會花費比較多的時間**。
- 透過使用 Rc::clone 來參考計數，可以以視覺辨別出這是深度拷貝的clone還是增加參考計數的clone。當需要調查程式碼的效能問題時，就只需要考慮深拷貝的clone，而不必在意 Rc::clone。

智慧指標

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("建立 a 後的計數 = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("建立 b 後的計數 = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("建立 c 後的計數 = {}", Rc::strong_count(&a));
    }
    println!("c 離開作用域後的計數 = {}", Rc::strong_count(&a));
}
```

clone Rc<T> 實例會增加其參考計數

- 改變之前範例，好能觀察參考計數在建立與釋放 a 的 Rc<List> 參考時產生的變化。
- 右上範例改變了 main 讓列表 c 寫在一個內部作用域中，然後就能觀察到當 c 離開作用域時參考計數產生的改變：
- 在程式中每次參考計數產生改變的地方，就印出參考計數，可以透過呼叫函式Rc::strong_count來取得。
- 此函式叫做 strong_count 而非 count 是因為 Rc<T> 型別還有個 weak_count，會在「避免參考循環：將 Rc<T> 轉換成 Weak<T>」段落看到 weak_count 的使用方式。

智慧指標

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.45s
     Running `target/debug/cons-list`
建立 a 後的計數 = 1
建立 b 後的計數 = 2
建立 c 後的計數 = 3
c 離開作用域後的計數 = 2
```

clone Rc<T> 實例會增加其參考計數

- 此程式碼印出右邊結果：
- 可以看到 a 的 Rc<List> 會有個初始參考計數 1，然後每次呼叫 clone 時，計數會加 1。當 c 離開作用域時，計數會減 1。不必呼叫任何函式來減少參考計數，像呼叫 Rc::clone 時才會增加參考計數那樣。
- 當 Rc<T> 數值離開作用域時，Drop 特徵的實作就會自動減少參考計數。
- 無法從此例觀察到的是當 b 然後是 a 從 main 的結尾離開作用域時，計數會是 0，然後 Rc<List> 在此時就會完全被清除。**使用 Rc<T> 能允許單一數值能有數個擁有者，然後計數會確保只要有任何擁有者還存在的狀況下，數值會保持是有效的**。

智慧指標

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.45s
     Running `target/debug/cons-list`
建立 a 後的計數 = 1
建立 b 後的計數 = 2
建立 c 後的計數 = 3
c 離開作用域後的計數 = 2
```

clone Rc<T> 實例會增加其參考計數

- 透過不可變參考，Rc<T> 能分享資料給程式中數個部分來只做讀取的動作。
- 如果 Rc<T> 允許也擁有數個可變參考的話，可能就違反了之前提及的借用規則：數個對相同位置的可變借用會導致資料競爭(data races)與不一致。但可變資料還是非常實用的！
- 在下個段落，會討論內部可變性模式與RefCell<T>型別，此型別能搭配 Rc<T> 使用來處理不可變的限制。

智慧指標

RefCell<T> 與內部可變性模式

- 內部可變性(Interior mutability)是 Rust 中的一種設計模式，能對即使是不可變參考的資料也能改變。正常狀況下，借用規則是不允許這種動作的。
- 為了改變資料，這樣的模式會在資料結構內使用 unsafe 程式碼來繞過 Rust 的常見可變性與借用規則。不安全(unsafe)的程式碼等於告訴編譯器會自己手動檢查，編譯器不會檢查全部的規則，會在之後討論更多關於不安全的程式碼。
- 當編譯器無法保障，但可以確保借用規則在執行時能夠遵循的話，就可以使用擁有內部可變性模式的型別。其內的 unsafe 程式碼會透過安全的 API 封裝起來，讓外部型別仍然是不可變的。
- 觀察擁有內部可變性模式的 RefCell<T> 型別來探討此概念。

智慧指標

透過 RefCell<T> 在執行時強制檢測借用規則

- 不像 Rc<T>，RefCell<T> 型別的資料只會有一個所有權。所以 RefCell<T> 與 Box<T> 這種型別有何差別呢？回憶一下在之前學到的借用規則：
  - 在任何時候，要麼只能有一個可變參考，要麼可以有任意數量的不可變參考。
  - 參考必須永遠有效。
- 對於參考與Box<T>，借用規則會在**編譯期**強制檢測。如果打破這些規則，會得到**編譯錯誤**。
- 對於 RefCell<T>，這些規則會在**執行時**才強制執行。如果打破這些規則，程式會**恐慌並離開**。
- 對於參考與Box<T>，而對 RefCell<T> 來說，在編譯時期檢查借用規則的優勢在於錯誤能在開發過程及早獲取，而且這對執行時的效能沒有任何影響，因為所有的分析都預先完成了。基於這些原因，在編譯時檢查借用規則在大多數情形都是最佳選擇，這也是為何這是 Rust 預設設置的原因。

智慧指標

透過 RefCell<T> 在執行時強制檢測借用規則

- 在執行時檢查借用規則的優勢則在於能允許一些特定記憶體安全的場合，而這些原本是不被編譯時檢查所允許的。像 Rust 編譯器這種靜態分析本質上是保守的。有些程式碼特性是無法透過分析程式碼檢測出的，最著名的範例就是**停機問題(Halting Problem)**。
- 因為有些分析是不可能的，如果 Rust 編譯器無法確定程式碼是否符合所有權規則，它可能會拒絕一支正確的程式，所以由此觀點來看能知道 Rust 編譯器是保守的。
- 如果 Rust 接受不正確的程式，使用者就無法信任 Rust 帶來的保障。然而如果 Rust 拒絕正確的程式，對程式設計師就會很不方便，但沒有任何嚴重的災難會發生。
- RefCell<T> 型別就適用於當確定程式碼有遵循借用規則，但編譯器無法理解並保證的時候。

智慧指標

透過 RefCell<T> 在執行時強制檢測借用規則

- 類似於 Rc<T>，RefCell<T> 也只能用於單一執行緒(single-threaded)的場合，如果嘗試用在多執行緒上的話就會出現編譯時錯誤。會在之後討論如何在多執行緒程式擁有 RefCell<T> 的功能。
- 以下是何時選擇 Box<T>、Rc<T> 或 RefCell<T> 的理由：
  - Rc<T> 讓**數個擁有者**能共享相同資料；Box<T> 與 RefCell<T> 只能有**一個擁有者**。
  - Box<T>能有**不可變或可變**的借用並在**編譯時檢查**；Rc<T> 則**只能有不可變**借用並在**編譯時檢查**：
  - RefCell<T> 能有**不可變或可變**借用但是在**執行時檢查**。
  - 由於 RefCell<T> 允許在執行時檢查可變參考，可以改變RefCell<T>內部的數值，就算RefCell<T>是不可變的。
- 改變不可變數值內部的值稱為內部可變性模式。看看內部可變性何時會有用，且觀察為何是可行的。

智慧指標

內部可變性：不可變數值的可變借用

- 借用規則的影響是當有個不可變數值，就無法取得可變參考。

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```

- 舉例來說，以下程式碼會無法編譯：
- 如果嘗試編譯此程式碼，會獲得以下錯誤：

```
$ cargo run
   Compiling borrowing v0.1.0 (file:///projects/borrowing)
error[E0596]: cannot borrow `x` as mutable, as it is not declared as mutable
 --> src/main.rs:3:13
  |
2 |     let x = 5;
  |         - help: consider changing this to be mutable: `mut x`
3 |     let y = &mut x;
  |             ^^^^^^ cannot borrow as mutable
```

- 然而在某些特定情況，會想要能夠有個方法可以改變一個數值，但該數值對其他程式碼而言仍然是不可變的。數值提供的方法以外的程式碼都無法改變其值。
- 使用 RefCell<T> 是取得內部可變性的方式之一。但 RefCell<T> 仍然要完全遵守借用規則：編譯器的借用檢查器會允許這些內部可變性，然後在執行時才檢查借用規則。如果違反規則，就會得到 panic! 而非編譯錯誤。
- 用一個實際例子來探討如何使用 RefCell<T> 來改變不可變數值，並瞭解為何這是很實用的。

智慧指標

內部可變性的使用案例：模擬物件

- 程式設計師有時在進行測試時會將一個型別替換成其他型別，用以觀察特定行為並判定是否有正確實作。這種型別就稱為測試替身(test double)。
- 可以想成這和影視產業中的「特技替身演員」類似，有個人會代替原本的演員來拍攝一些特定的場景。測試替身會在執行測試時代替其他型別。模擬物件(Mock objects)是測試替身其中一種特定型別，這能紀錄測試過程中發生什麼事並能判斷動作是否正確。
- Rust 的物件與其他語言中的物件概念並不全然相同，而且 Rust 的標準函式庫內也沒有如其他語言會內建的模擬物件功能。不過還是可以有方法來建立結構體來作為模擬物件。
- 以下是要測試的情境：建立一個函式庫來追蹤一個**數值與最大值的差距**，並依據該差距傳送訊息。舉例來說，此函式庫就能用來追蹤使用者允許呼叫 API 次數的上限。

智慧指標

內部可變性的使用案例：模擬物件

- 函式庫提供的功能只有追蹤與最大值的距離以及何時該傳送什麼訊息。
- 使用函式庫的應用程式要提供傳送訊息的機制，應用程式可以將訊息存在應用程式內、傳送電子郵件、傳送文字訊息或其他等等。
- 函式庫不需要知道細節，它只需要在意會有項目實作提供的 Messenger 特徵。右邊範例顯示了函式庫的程式碼：

```rust
pub trait Messenger {
    fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
where
    T: Messenger,
{
    pub fn new(messenger: &'a T, max: usize) -> LimitTracker<'a, T> {
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }

    pub fn set_value(&mut self, value: usize) {
        self.value = value;

        let percentage_of_max = self.value as f64 / self.max as f64;

        if percentage_of_max >= 1.0 {
            self.messenger.send("錯誤：你超過使用上限了！");
        } else if percentage_of_max >= 0.9 {
            self.messenger
                .send("緊急警告：你已經使用 90% 的配額了！");
        } else if percentage_of_max >= 0.75 {
            self.messenger
                .send("警告：你已經使用 75% 的配額了！");
        }
    }
}
```

智慧指標

內部可變性的使用案例：模擬物件

- 此程式碼其中一個重點是 Messenger 特徵有個方法叫做 send，這會接收一個 self 的不可變參考與一串訊息文字。
- 此特徵就是模擬物件所需實作的介面，能模擬和實際物件一樣的行爲。
- 另一個重點是想要測試 LimitTracker 中 set_value 方法的行為。可以改變傳給參數 value 的值，但是 set_value 沒有回傳任何東西好做判斷。
- 希望如果透過某個實作 Messenger 的型別與特定數值 max 來建立 LimitTracker 時，傳送訊息者能被通知要傳遞合適的訊息。

```rust
pub trait Messenger {
    fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
where
    T: Messenger,
{
    pub fn new(messenger: &'a T, max: usize) -> LimitTracker<'a, T> {
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }

    pub fn set_value(&mut self, value: usize) {
        self.value = value;

        let percentage_of_max = self.value as f64 / self.max as f64;

        if percentage_of_max >= 1.0 {
            self.messenger.send("錯誤：你超過使用上限了！");
        } else if percentage_of_max >= 0.9 {
            self.messenger
                .send("緊急警告：你已經使用 90% 的配額了！");
        } else if percentage_of_max >= 0.75 {
            self.messenger
                .send("警告：你已經使用 75% 的配額了！");
        }
    }
}
```

智慧指標

內部可變性的使用案例：模擬物件

- 需要有個模擬物件，而不是在呼叫 send 時真的傳送電子郵件或文字訊息，只想紀錄訊息被通知要傳送了。
- 可以建立模擬物件的實例，以此建立LimitTracke、呼叫LimitTracker的 set_value，並檢查模擬物件有預期的訊息。
- 右邊範例展示一個嘗試實作的模擬物件，但**借用檢查器卻不允許**：

```rust
#[cfg(test)]
mod tests {
    use super::*;

    struct MockMessenger {
        sent_messages: Vec<String>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: vec![],
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        let mock_messenger = MockMessenger::new();
        let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);

        limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_messages.len(), 1);
    }
}
```

智慧指標

內部可變性的使用案例：模擬物件

- 此測試程式碼定義了一個結構體 MockMessenger 其有個 sent_messages 欄位並存有 String 數值的 Vec 來追蹤被通知要傳送的訊息。
- 也定義了一個關聯函式 new 可以方便建立起始訊息列表為空的 MockMessenger。
- 對 MockMessenger 實作 Messenger 特徵，這樣才能將 MockMessenger 交給 LimitTracker。
- 在 send 方法的定義中，取得由參數傳遞的訊息，並存入 MockMessenger 的 sent_messages 列表中。

```rust
#[cfg(test)]
mod tests {
    use super::*;

    struct MockMessenger {
        sent_messages: Vec<String>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: vec![],
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        let mock_messenger = MockMessenger::new();
        let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);

        limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_messages.len(), 1);
    }
}
```

智慧指標

內部可變性的使用案例：模擬物件

- 在測試中，測試當 LimitTracker 被通知將 value 設為超過 max 數值 75% 的某個值。
- 首先，建立新的 MockMessenger，其起始為一個空的訊息列表。然後建立一個新的 LimitTracker 並將 MockMessenger 的參考與一個 max 為 100 的數值賦值給它。
- 用數值 80 來呼叫 LimitTracker 的 set_value 方法此值會超過 100 的 75%。
- 然後判定 MockMessenger 追蹤的訊息列表需要至少有一個訊息。

```
#[cfg(test)]
mod tests {
    use super::*;

    struct MockMessenger {
        sent_messages: Vec<String>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: vec![],
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        let mock_messenger = MockMessenger::new();
        let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);

        limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_messages.len(), 1);
    }
}
```

智慧指標

內部可變性的使用案例：模擬物件

- 但是此測試有個問題，如以下所示：

```
$ cargo test
   Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
error[E0596]: cannot borrow `self.sent_messages` as mutable, as it is behind a `&` reference
  --> src/lib.rs:58:13
   |
2  |     fn send(&self, msg: &str);
   |             ----- help: consider changing that to be a mutable reference: `&mut self`
...
58 |             self.sent_messages.push(String::from(message));
   |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `self` is a `&` reference, so the data
   it refers to cannot be borrowed as mutable
```

- 無法修改 MockMessenger 來追蹤訊息，**因為 send 方法取得的是 self 的不可變參考**。而也無法使用錯誤訊息中推薦使用的 &mut self，因為 send 的簽名就會與 Messenger 特徵所定義的不相符(可以試看看並觀察錯誤訊息)。

智慧指標

內部可變性的使用案例：模擬物件

- 這就是內部可變性能帶來幫助的場合！會將 sent_messages 存入 RefCell<T> 內，然後 send 方法就也能夠進行修改存入訊息。右邊範例顯示了變更後的程式碼：
- sent_messages 欄位現在是型別 RefCell<Vec<String>> 而非 Vec<String>。
- 在 new 函式中，用空的向量來建立新的 RefCell<Vec<String>>。

```
#[cfg(test)]
mod tests {
    use super::*;
    use std::cell::RefCell;

    struct MockMessenger {
        sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: RefCell::new(vec![]),
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.borrow_mut().push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        // --省略--

        assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}
```

智慧指標

內部可變性的使用案例：模擬物件

- 至於 send 方法的實作，第一個參數仍然是 self 的不可變借用，這就符合特徵所定義的。
- 在 self.sent_messages 對 RefCell<Vec<String>> **呼叫 borrow_mut 來取得 RefCell<Vec<String>> 內的可變參考數值，也就是向量**。然後對向量的可變參考呼叫 push 來追蹤測試中的訊息。
- 最後一項改變是判定：要看到內部向量有多少項目的話，對 RefCell<Vec<String>> 呼叫 borrow 來取得向量的不可變參考。
- 現在已經知道如何使用 RefCell<T>，進一步探討它如何運作的吧！

```
#[cfg(test)]
mod tests {
    use super::*;
    use std::cell::RefCell;

    struct MockMessenger {
        sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: RefCell::new(vec![]),
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.borrow_mut().push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        // --省略--

        assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}
```

智慧指標

透過 RefCell<T> 在執行時追蹤借用

- 當建立不可變與可變參考時，分別使用 & 和 &mut 語法。而對於 RefCell<T> 的話，使用 **borrow** 和 **borrow_mut** 方法，這是 RefCell<T> 所提供的安全 API 之一。
  - borrow 方法回傳一個智慧指標型別 Ref<T>；而 borrow_mut 回傳智慧指標型別 RefMut<T>。
  - 這兩個型別都有實作 Deref，所以可以像一般參考來對待它們。
- **RefCell<T> 會追蹤當前有多少 Ref<T> 和 RefMut<T> 智慧指標存在**。每次呼叫 borrow 時，RefCell<T> 會增加不可變借用計數。當 Ref<T> 離開作用域時，不可變借用計數就會減一。就和編譯時借用規則一樣，**RefCell<T> 同一時間要麼只能有一個可變參考，要麼可以有數個不可變參考。**

智慧指標

```
impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        let mut one_borrow = self.sent_messages.borrow_mut();
        let mut two_borrow = self.sent_messages.borrow_mut();

        one_borrow.push(String::from(message));
        two_borrow.push(String::from(message));
    }
}
```

透過 RefCell<T> 在執行時追蹤借用

- 如果嘗試違反這些規則，不會像參考那樣得到編譯器錯誤，RefCell<T> 的實作會在執行時恐慌。右上範例修改了之前範例的send實作。故意嘗試在同個作用域下建立兩個可變參考，來說明 RefCell<T> 會不允許執行時這樣做：
- 從 borrow_mut 回傳的 RefMut<T> 智慧指標來建立變數 one_borrow。然後再以相同方式建立另一個變數 two_borrow。這在同個作用域下產生了兩個可變參考，而這是不允許的。

智慧指標

```
impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        let mut one_borrow = self.sent_messages.borrow_mut();
        let mut two_borrow = self.sent_messages.borrow_mut();

        one_borrow.push(String::from(message));
        two_borrow.push(String::from(message));
    }
}
```

透過 RefCell<T> 在執行時追蹤借用

- 執行函式庫的測試時，右上範例可以編譯通過，但是執行測試會失敗：

```
$ cargo test
   Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
    Finished test [unoptimized + debuginfo] target(s) in 0.91s
     Running unittests src/lib.rs (target/debug/deps/limit_tracker-e599811fa246dbde)

running 1 test
test tests::it_sends_an_over_75_percent_warning_message ... FAILED

failures:

---- tests::it_sends_an_over_75_percent_warning_message stdout ----
thread 'main' panicked at 'already borrowed: BorrowMutError', src/lib.rs:60:53
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::it_sends_an_over_75_percent_warning_message

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

error: test failed, to rerun pass `--lib`
```

智慧指標

```
$ cargo test
   Compiling limit-tracker v0.1.0 (file:///projects/limit-tracker)
    Finished test [unoptimized + debuginfo] target(s) in 0.91s
     Running unittests src/lib.rs (target/debug/deps/limit_tracker-e599811fa246dbde)

running 1 test
test tests::it_sends_an_over_75_percent_warning_message ... FAILED

failures:

---- tests::it_sends_an_over_75_percent_warning_message stdout ----
thread 'main' panicked at 'already borrowed: BorrowMutError', src/lib.rs:60:53
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    tests::it_sends_an_over_75_percent_warning_message

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

error: test failed, to rerun pass `--lib`
```

透過 RefCell<T> 在執行時追蹤借用

- 注意到程式碼恐慌時的訊息 already borrowed: BorrowMutError。這就是 RefCell<T> 如何在執行時處理違反借用規則的情況。
- 像在這裡選擇在執行時獲取借用錯誤而不是在編譯時，代表會在開發過程之後才找到程式碼錯誤，甚至有可能一直到程式碼部署到正式環境後才查覺。而且程式碼也會多了一些小小的執行時效能開銷，作為在執行時而非編譯時檢查的代價。
- 不過使用 RefCell<T> 能在只允許有不可變數值的環境中寫出能夠變更內部追蹤訊息的模擬物件。這是想獲得 RefCell<T> 帶來的功能時，要與一般參考之間作出的取捨。

智慧指標

組合 Rc<T> 與 RefCell<T> 來擁有多個可變資料的擁有者

- RefCell<T> 的常見使用方法是搭配 Rc<T>。回想一下 Rc<T> 可以對數個擁有者共享相同資料，但是它只能用於不可變資料。如果有一個 Rc<T> 並存有 RefCell<T> 的話就可以取得一個有數個擁有者而且可變的數值！
- 舉例來說，回憶一下之前範例cons list 的範例使用了 Rc<T> 來讓數個列表可以共享另一個列表的所有權。因為Rc<T>只能有不可變數值，一旦建立它們後就無法變更列表中的任何數值。加上RefCell<T>來獲得能改變列表數值的能力。
- 右邊範例顯示了在Cons定義中使用 RefCell<T>，這樣一來就可以變更儲存在列表中的所有數值：

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a 之後 = {:?}", a);
    println!("b 之後 = {:?}", b);
    println!("c 之後 = {:?}", c);
}
```

智慧指標

組合 Rc<T> 與 RefCell<T> 來擁有多個可變資料的擁有者

- 建立了一個 Rc<RefCell<i32>> 實例數值並將其存入變數 value 好讓之後可以直接取得。
- 然後在 a 用持有 value 的 Cons 變體來建立 List。需要clone value，這樣 a 和 value 才能都有內部數值 5 的所有權，而不是從 value 轉移所有權給 a，或是讓 a 借用 value。
- 用 Rc<T> 封裝列表 a，所以當建立列表 b 和 c 時，它們都可以參考 a，就像之前範例一樣。

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a 之後 = {:?}", a);
    println!("b 之後 = {:?}", b);
    println!("c 之後 = {:?}", c);
}
```

智慧指標

組合 Rc<T> 與 RefCell<T> 來擁有多個可變資料的擁有者

- 在建立完列表a、b 和 c 之後，想對value的數值加上 10。對 value 呼叫 borrow_mut，其中使用到了之前討論過的自動解參考功能來解參考Rc<T>成內部的 RefCell<T>數值。
- borrow_mut 方法會回傳 RefMut<T> 智慧指標，而使用解參考運算子並改變其內部數值。

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a 之後 = {:?}", a);
    println!("b 之後 = {:?}", b);
    println!("c 之後 = {:?}", c);
}
```

智慧指標

組合 Rc<T> 與 RefCell<T> 來擁有多個可變資料的擁有者

- 當印出 a、b 和 c 時，可以看到它們的數值都改成了 15 而非 5：

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.63s
     Running `target/debug/cons-list`
a 之後 = Cons(RefCell { value: 15 }, Nil)
b 之後 = Cons(RefCell { value: 3 }, Cons(RefCell { value: 15 }, Nil))
c 之後 = Cons(RefCell { value: 4 }, Cons(RefCell { value: 15 }, Nil))
```

- 透過使用 RefCell<T>，可以得到一個外部是不可變的 List 數值，但是可以使用 RefCell<T> 提供的方法來取得其內部可變性，可以在想要時改變資料。執行時的借用規則檢查能防止資料競爭，並在某些場合犧牲一點速度來換取資料結構的彈性。
- 注意到 RefCell<T> 無法用在多執行緒的程式碼！Mutex<T> 才是執行緒安全版的 RefCell<T>，會在之後再討論 Mutex<T>。

智慧指標

參考循環會導致記憶體泄漏

- 意外情況下，執行程式時可能會產生永遠不會被清除的記憶體(通稱為記憶體泄漏 / memory leak)。Rust 的記憶體安全性雖然可以保證令這種情況難以發生，但並非絕不可能。
- 雖然 Rust 在編譯時可以保證做到禁止資料競爭(data races)，但它無法保證完全避免記憶體泄漏，這是因為對 Rust 來說，記憶體泄漏是屬於安全範疇內的(memory safe)。
- 透過使用 Rc<T> 和 RefCell<T> ，能觀察到 Rust 允許使用者自行產生記憶體泄漏：**因為使用者可以產生兩個參考並互相參照，造成一個循環**。這種情況下會導致記憶體泄漏，因為循環中的參考計數永遠不會變成 0，所以數值永遠不會被釋放。

智慧指標

產生參考循環

- 看看參考循環是怎麼發生的，以及如何避免它。從底下範例的 List 列舉定義與一個 tail 方法開始：

```
use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match self {
            Cons(_, item) => Some(item),
            Nil => None,
        }
    }
}

fn main() {}
```

- 用的是之前範例中 List 的另一種定義寫法。
- Cons 變體的第二個元素現在是 RefCell<Rc<List>>，代表不同於之前範例那樣能修改 i32 數值，想要能修改 Cons 變體指向的 List 數值。
- 也加了一個 tail 方法，如果有 Cons 變體的話，能方便取得第二個項目。

智慧指標

產生參考循環

- 在右邊範例加入 main 函式並使用之前範例的定義。此程式碼建立了列表 a 與指向列表 a 的列表 b。然後它修改了列表 a 來指向 b，因而產生循環參考。在程序過程中 println! 陳述式會顯示不同位置時的參考計數。

```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a 初始參考計數 = {}", Rc::strong_count(&a));
    println!("a 下個項目 = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a 在 b 建立後的參考計數 = {}", Rc::strong_count(&a));
    println!("b 初始參考計數 = {}", Rc::strong_count(&b));
    println!("b 下個項目 = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b 在變更 a 後的參考計數 = {}", Rc::strong_count(&b));
    println!("a 在變更 a 後的參考計數 = {}", Rc::strong_count(&a));

    // 取消下一行的註解可以看到循環產生
    // 這會讓堆疊溢位
    // println!("a 下個項目 = {:?}", a.tail());
}
```

智慧指標

產生參考循環

- 在變數 a 建立了一個 Rc<List> 實例的 List 數值並持有 5, Nil 初始列表的。然後在變數 b 建立另一個 Rc<List> 實例的 List 數值並持有數值 10 與指向的列表 a。
- 將 a 修改為指向 b 而非 Nil 來產生循環。透過使用 tail 方法來取得 a 的 RefCell<Rc<List>> 參考，並放入變數 link 中。
- 然後對 RefCell<Rc<List>> 使用 borrow_mut 方法來改變 Rc<List> 的值，從數值 Nil 改成 b 的 Rc<List>。

```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a 初始參考計數 = {}", Rc::strong_count(&a));
    println!("a 下個項目 = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a 在 b 建立後的參考計數 = {}", Rc::strong_count(&a));
    println!("b 初始參考計數 = {}", Rc::strong_count(&b));
    println!("b 下個項目 = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b 在變更 a 後的參考計數 = {}", Rc::strong_count(&b));
    println!("a 在變更 a 後的參考計數 = {}", Rc::strong_count(&a));

    // 取消下一行的註解可以看到循環產生
    // 這會讓堆疊溢位
    // println!("a 下個項目 = {:?}", a.tail());
}
```

智慧指標

產生參考循環

- 當執行此程式並維持將最後一行的 println! 註解掉的話，會得到以下輸出：

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.53s
     Running `target/debug/cons-list`
a 初始參考計數 = 1
a 下個項目 = Some(RefCell { value: Nil })
a 在 b 建立後的參考計數 = 2
b 初始參考計數 = 1
b 下個項目 = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
b 在變更 a 後的參考計數 = 2
a 在變更 a 後的參考計數 = 2
```

```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a 初始參考計數 = {}", Rc::strong_count(&a));
    println!("a 下個項目 = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a 在 b 建立後的參考計數 = {}", Rc::strong_count(&a));
    println!("b 初始參考計數 = {}", Rc::strong_count(&b));
    println!("b 下個項目 = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b 在變更 a 後的參考計數 = {}", Rc::strong_count(&b));
    println!("a 在變更 a 後的參考計數 = {}", Rc::strong_count(&a));

    // 取消下一行的註解可以看到循環產生
    // 這會讓堆疊溢位
    // println!("a 下個項目 = {:?}", a.tail());
}
```

智慧指標

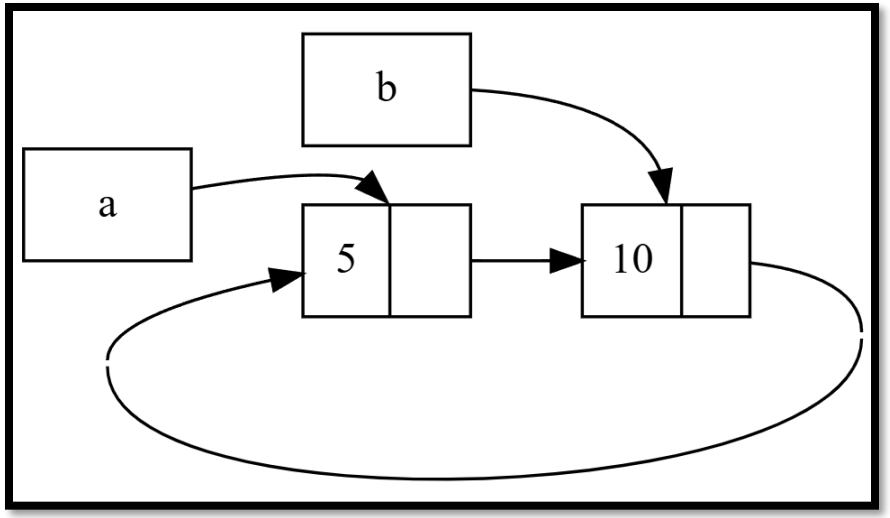
產生參考循環

```rust
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a 初始參考計數 = {}", Rc::strong_count(&a));
    println!("a 下個項目 = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a 在 b 建立後的參考計數 = {}", Rc::strong_count(&a));
    println!("b 初始參考計數 = {}", Rc::strong_count(&b));
    println!("b 下個項目 = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b 在變更 a 後的參考計數 = {}", Rc::strong_count(&b));
    println!("a 在變更 a 後的參考計數 = {}", Rc::strong_count(&a));

    // 取消下一行的註解可以看到循環產生
    // 這會讓堆疊溢位
    // println!("a 下個項目 = {:?}", a.tail());
}
```

- 在變更列表 a 來指向 b 後，a 和 b 的 Rc<List> 實例參考計數都是 2。在 main 結束後，Rust 會釋放 b，讓 b 的 Rc<List> 實例計數從 2 減到 1。此時堆積上 Rc<List> 的記憶體還不會被釋放，因爲參考計數還有 1，而非 0。
- 然後 Rust 釋放 a，讓 a 的 Rc<List> 實例也從 2 減到 1。此實例的記憶體也不會被釋放，因爲另一個 Rc<List> 的實例仍然參考著它。

```
$ cargo run
   Compiling cons-list v0.1.0 (file:///projects/cons-list)
    Finished dev [unoptimized + debuginfo] target(s) in 0.53s
     Running `target/debug/cons-list`
a 初始參考計數 = 1
a 下個項目 = Some(RefCell { value: Nil })
a 在 b 建立後的參考計數 = 2
b 初始參考計數 = 1
b 下個項目 = Some(RefCell { value: Cons(5, RefCell { value: Nil }) })
b 在變更 a 後的參考計數 = 2
a 在變更 a 後的參考計數 = 2
```

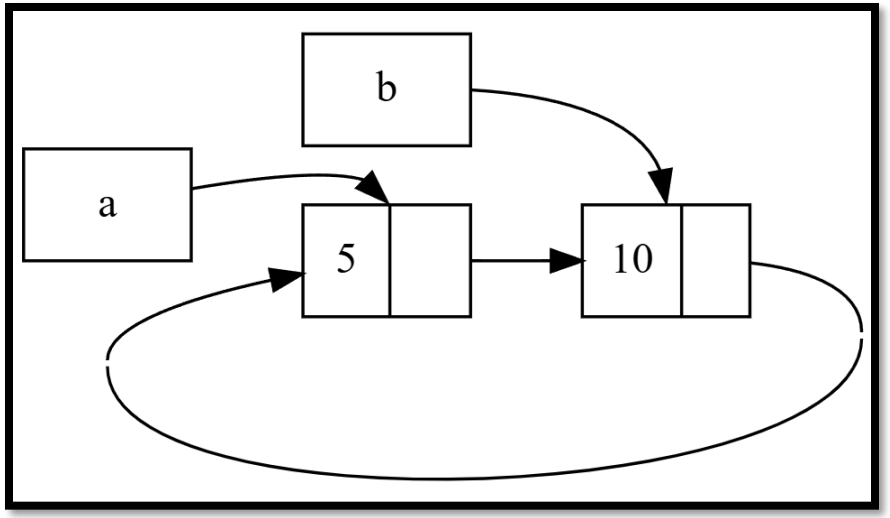
智慧指標



產生參考循環

- 列表配置的記憶體會永遠不被釋放。為了視覺化參考循環，用右上圖示表示：
- 如果解除最後一個 println! 的註解並執行程式的話，Rust 會嘗試印出此循環，因為 a 會指向 b 會指向 a 以此循環下去，直到堆疊溢位(stack overflow)。
- 比起真實世界的程式，此循環造成影響並不嚴重。因為當建立完循環參考，程式就結束了。不過要是有個更複雜的程式配置了大量的記憶體而產生循環，並維持很長一段時間的話，程式會用到比原本預期還多的記憶體，並可能壓垮系統，導致它將記憶體用光。

智慧指標



產生參考循環

- 要產生循環參考並不是件容易的事，但也不是絕對不可能。如果有包含 Rc<T> 數值的 RefCell<T> 數值，或是有類似具內部可變性與參考計數巢狀組合的話，必須確保不會產生循環參考，無法依靠 Rust 來檢查它們。產生循環參考是程式中的邏輯錯誤，需要使用自動化測試、程式碼審查以及其他軟體開發技巧來最小化問題。
- 另一個避免參考循環的解決辦法是重新組織資料結構，確定哪些參考要有所有權，哪些參考不用。這樣一來，循環會由一些有所有權的關係與沒有所有權的關係所組成，而只有所有權關係能影響數值是否能被釋放。
- 在之前範例中，永遠會希望 Cons 變體擁有它們的列表，所以重新組織資料結構是不可能的。看看一個由父節點與子節點組成的圖形結構，來看看無所有權的關係何時適合用來避免循環參考。

智慧指標

避免參考循環：將 Rc<T> 轉換成 Weak<T>

- 目前解釋過呼叫 Rc::clone 會增加 Rc<T> 實例的 strong_count，**而 Rc<T> 只會在 strong_count 為 0 時被清除。**
- 也可以對 Rc<T> 實例呼叫 Rc::downgrade 並傳入 Rc<T> 的參考來建立弱參考(weak reference)。強參考是分享 Rc<T> 實例的方式。**弱參考不會表達所有權關係，它們的計數與 Rc<T> 的清除無關。它們不會造成參考循環，因為弱參考的循環會在其強參考計數歸零時解除。**
- 當呼叫 Rc::downgrade 時，會得到一個型別為 Weak<T> 的智慧指標。不同於對 Rc<T> 實例的 strong_count 增加 1，呼叫 Rc::downgrade 會對 weak_count 增加 1。
- Rc<T> 型別使用 weak_count 來追蹤有多少 Weak<T> 的參考存在，這類似於 strong_count。**不同的地方在於 weak_count 不需要歸零才能將 Rc<T> 清除。**

智慧指標

避免參考循環：將 Rc<T> 轉換成 Weak<T>

- 由於 Weak<T> 的參考數值可能會被釋放，要對 Weak<T> 指向的數值做任何事情時，都必須確保該數值還存在。
- 可以透過對 Weak<T> 實例呼叫 upgrade 方法，這會回傳 Option<Rc<T>>。如果 Rc<T> 數值還沒被釋放的話，就會得到 Some；而如果 Rc<T> 數值已經被釋放的話，就會得到 None。因為 upgrade 回傳 Option<Rc<T>>，Rust 會確保 Some 與 None 的分支都有處理好，所以不會取得無效指標。
- 為了做示範，與其使用知道下一項的列表的例子，會建立一個樹狀結構，每一個項目會知道它們的子項目以及它們的父項目。

智慧指標

建立樹狀資料結構：帶有子節點的 Node

- 首先建立一個帶有節點的樹，每個節點知道它們的子節點。會定義一個結構體 Node 來存有它自己的 i32 數值以及其子數值 Node 的參考：

```
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
struct Node {
    value: i32,
    children: RefCell<Vec<Rc<Node>>>,
}
```

- 想要Node擁有自己的子節點，而且想要透過變數分享所有權，可以在樹中取得每個 Node。為此定義 Vec<T> 項目作為型別 Rc<Node> 的數值。還想要能夠修改哪些節點才是該項目的子節點，所以將 children 中的 Vec<Rc<Node>> 加進 RefCell<T>。

智慧指標

建立樹狀資料結構：帶有子節點的 Node

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        children: RefCell::new(vec![]),
    });

    let branch = Rc::new(Node {
        value: 5,
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });
}
```

- 接著使用定義的結構體來建立一個 Node 實例叫做 leaf，其數值為 3 且沒有子節點；再建立另一個實例叫做 branch，其數值為 5 且有個子節點 leaf。如右上範例所示：
- clone leaf 的 Rc<Node> 並存入branch，代表leaf的 Node 現在有兩個擁有者：leaf 和 branch。可以透過 branch.children 從 branch 取得 leaf，但是從 leaf 無法取得 branch。
- 原因是因為 leaf 沒有 branch 的參考且不知道它們之間是有關聯的。想要 leaf 能知道 branch 是它的父節點。這就是接下來要做的事。

智慧指標

新增從子節點到父節點的參考

- 要讓子節點意識到它的父節點，需要在 Node 結構體定義中加個 parent 欄位。
- 問題在於 parent 應該要是什麼型別。知道它不能包含 Rc<T>，因為那就會造成參考循環，leaf.parent 就會指向 branch 且 branch.children 就會指向 leaf，導致同名的 strong_count 數值無法歸零。
- 換種方式思考此關係，父節點必須擁有它的子節點，如果父節點釋放的話，它的子節點也應該要被釋放。但子節點不應該擁有它的父節點，如果釋放子節點的話，父節點應該要還存在。這就是弱參考的使用時機！

智慧指標

新增從子節點到父節點的參考

```rust
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}
```

- 所以與其使用 Rc<T>，使用 Weak<T> 來建立 parent 的型別，更明確的話就是 RefCell<Weak<Node>>。現在 Node 結構體定義看起來會像這樣：
- 子節點能夠參考其父節點但不會擁有它。在底下範例中更新了 main 來使用新的定義，讓 leaf 節點有辦法參考它的父節點 branch：

```rust
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf 的父節點 {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf 的父節點 {:?}", leaf.parent.borrow().upgrade());
}
```

智慧指標

```rust
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf 的父節點 {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf 的父節點 {:?}", leaf.parent.borrow().upgrade());
}
```

新增從子節點到父節點的參考

- 建立 leaf 節點與之前範例類似，只是要多加個 parent 欄位：leaf 一開始沒有任何父節點，所以建立一個空的 Weak<Node> 參考實例。
- 此時當透過 upgrade 方法嘗試取得 leaf 的父節點參考的話，會取得 None 數值。能在輸出結果的第一個 println! 陳述式看到：`leaf 的父節點 None`
- 當建立 branch 節點，它的 parent 欄位也會有個新的 Weak<Node> 參考，因為 branch 沒有父節點。仍然有 leaf 作為 branch 其中一個子節點。一旦有了 branch 的 Node 實例，可以修改 leaf 使其擁有父節點的 Weak<Node> 參考。
- 對leaf中parent欄位的RefCell<Weak<Node>>使用 borrow_mut 方法，然後使用 Rc::downgrade 函式來從 branch 的 Rc<Node> 建立一個 branch 的 Weak<Node> 參考。

智慧指標

新增從子節點到父節點的參考

- 當再次印出 leaf 的父節點，這次就會取得 Some 變體其內就是 branch，現在 leaf 可以取得它的父節點了！當印出 leaf，也能避免產生像之前範例那樣最終導致堆疊溢位(stack overflow)的循環，Weak<Node> 會印成 (Weak)：

```
leaf 的父節點 Some(Node { value: 5, parent: RefCell { value: (Weak) },
children: RefCell { value: [Node { value: 3, parent: RefCell { value: (Weak) },
children: RefCell { value: [] } }] } })
```

- **沒有無限的輸出代表此程式碼沒有產生參考循環。**
- 也能透過呼叫Rc::strong_count與 Rc::weak_count 的數值看出。

智慧指標

視覺化 strong_count 與 weak_count 的變化

- 看看 Rc<Node> 實例中 strong_count 與 weak_count 的數值如何變化，建立一個新的內部作用域，並將branch的產生移入作用域中。
- 這樣就能看到branch建立與離開作用域而釋放時發生了什麼事。如右邊範例所示：

```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!(
        "leaf 的強參考 = {}、弱參考 = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
        });

        *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

        println!(
            "branch 的強參考 = {}、弱參考 = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );

        println!(
            "leaf 的強參考 = {}、弱參考 = {}",
            Rc::strong_count(&leaf),
            Rc::weak_count(&leaf),
        );
    }

    println!("leaf 的父節點 {:?}", leaf.parent.borrow().upgrade());
    println!(
        "leaf 的強參考 = {}、弱參考 = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}
```

智慧指標

視覺化 strong_count 與 weak_count 的變化

- 在 leaf 建立後，它的 Rc<Node> 有**強計數為 1 與弱計數為 0**。
- 在內部作用域中，建立了 branch 並與 leaf 做連結，此時當印出計數時，branch的 Rc<Node> 會有強計數為1與弱計數為 1(因為 leaf.parent 透過 Weak<Node> 指向 branch)。
- 當印出 leaf 的計數時，會看到它會有強計數為 2，因為 branch 現在有個 leaf 的 Rc<Node> clone儲存在 branch.children，但弱計數仍為 0。
- **當內部作用域結束時，branch 會離開作用域且 Rc<Node> 的強計數會歸零，所以它的 Node 就會被釋放。leaf.parent 的弱計數 1 與 Node 是否被釋放無關，所以沒有產生任何記憶體泄漏！**

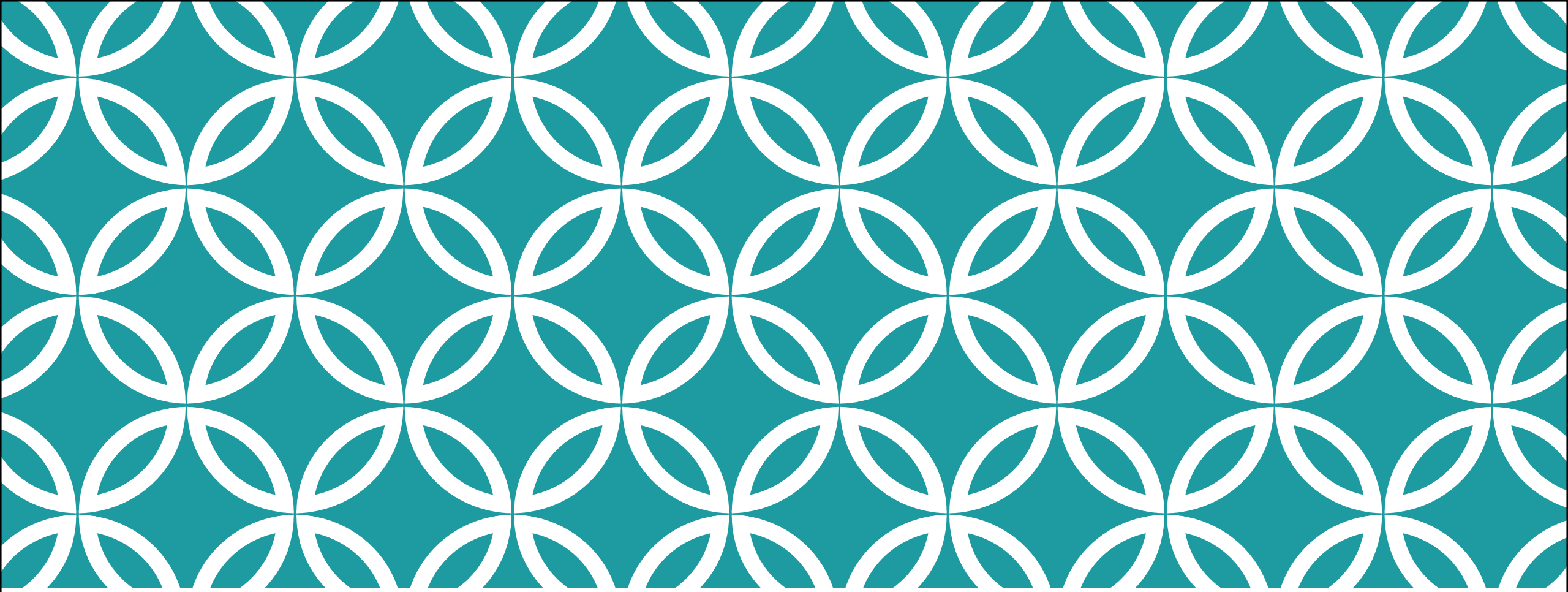
```
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!(
        "leaf 的強參考 = {}、弱參考 = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
        });

        *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

        println!(
            "branch 的強參考 = {}、弱參考 = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );

        println!(
            "leaf 的強參考 = {}、弱參考 = {}",
            Rc::strong_count(&leaf),
            Rc::weak_count(&leaf),
        );
    }

    println!("leaf 的父節點 {:?}", leaf.parent.borrow().upgrade());
    println!(
        "leaf 的強參考 = {}、弱參考 = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}
```

智慧指標

視覺化 strong_count 與 weak_count 的變化

- 如果嘗試在作用域結束後取得 leaf 的父節點，會再次獲得 None。
- 在程式的最後，leaf 的Rc<Node>強計數為1且弱計數為 0，因為變數 leaf 現在是 Rc<Node> 唯一的參考。
- 所有管理計數與數值釋放都已經實作在 Rc<T> 與 Weak<T>，它們都有 Drop 特徵的實作。
- 在 Node 的定義中指定子節點對父節點的關係應為 Weak<T> 參考，能夠將父節點與子節點彼此關聯，且不必擔心產生參考循環與記憶體泄漏。

```rust
fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!(
        "leaf 的強參考 = {}、弱參考 = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );

    {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
        });

        *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

        println!(
            "branch 的強參考 = {}、弱參考 = {}",
            Rc::strong_count(&branch),
            Rc::weak_count(&branch),
        );

        println!(
            "leaf 的強參考 = {}、弱參考 = {}",
            Rc::strong_count(&leaf),
            Rc::weak_count(&leaf),
        );
    }

    println!("leaf 的父節點 {:?}", leaf.parent.borrow().upgrade());
    println!(
        "leaf 的強參考 = {}、弱參考 = {}",
        Rc::strong_count(&leaf),
        Rc::weak_count(&leaf),
    );
}
```



無懼並行

無懼並行

能夠安全高效處理並行程式設計是Rust的另一項主要目標。**並行程式設計(Concurrent programming)會讓程式的不同部分獨立執行，而平行程式設計(parallel programming)則是程式的不同部分同時執行**。這些隨著電腦越能善用多處理器時也越顯得重要。歷史上，這種程式設計是很困難且容易出錯的，Rust 希望能改善這點。

起初 Rust 團隊認為**確保記憶體安全與預防並行問題**是兩個分別的問題，要用不同的解決方案。隨著時間過去，團隊發現**所有權與型別系統**同時是管理記憶體安全以及並行問題的強大工具！透過藉助所有權與型別檢查，許多並行錯誤在 Rust 中都是編譯時錯誤而非執行時錯誤。

因此，不用花大量時間嘗試重現編譯時並行錯誤出現時的特定情況，不正確的程式碼會在編譯時就被拒絕，並顯示錯誤解釋問題原因。這樣一來，就可以在開發時就修正問題，而不用等到可能都部署到生產環境了才發現問題。稱呼這個 Rust 的特色為**無懼並行(fearless concurrency)**。無懼並行可以避免寫出有微妙錯誤的程式碼，並能輕鬆重構，不用擔心產生新的程式錯誤。

無懼並行

許多語言對於處理並行問題所提供的解決方案都很有特色。舉例來說，Erlang 有非常優雅的訊息傳遞並行功能，但跨執行緒共享狀態就只有比較隱晦的方法。只提供支援可能解決方案的子集對於高階語言來說是合理的策略，因為高階語言所承諾的效益來自於犧牲一些掌控以換取大量的抽象層面。

然而，低階語言則預期會提供在任何給定場合中能有最佳效能的解決方案，而且對硬體的抽象較少。因此 Rust 提供了多種工具來針對適合場合與需求將問題定義出來。

本段落中會涵蓋這些主題：

- 如何建立執行緒(threads)來同時執行多段程式碼
- 訊息傳遞(Message-passing)並行提供通道(channels)在執行緒間傳遞訊息
- 共享狀態(Shared-state)並行提供多執行緒可以存取同一位置的資料
- Sync 與 Send 特徵擴展 Rust 的並行保障至使用者定義的型別與標準函式庫的型別中

無懼並行

使用執行緒同時執行程式碼

- 在大部分的現代作業系統中，被執行的程式碼會在程序(process)中執行，作業系統會負責同時處理數個程序。在程式中，也可以將各自獨立的部分同時執行。執行這些獨立部分的功能就叫做執行緒(threads)。舉例來說，一個網路伺服器可以有數個執行緒來同時回應一個以上的請求。
- 將程式中的運算拆成數個執行緒可以提升效能，不過這也同時增加了複雜度。因為執行緒可以同時執行，所以無法保證不同執行緒的程式碼執行的順序。這會導致以下問題：
  - 競爭條件(Race conditions)：數個執行緒以不一致的順序取得資料或資源
  - 死結(Deadlocks)：兩個執行緒彼此都在等待對方，因而讓執行緒無法繼續執行
  - 只在特定情形會發生的程式錯誤，並難以重現與穩定修復

無懼並行

使用執行緒同時執行程式碼

- Rust 嘗試降低使用執行緒所帶來的負面效果，不過對於多執行緒程式設計還是得格外小心，其所要求的程式結構也與單一執行緒的程式有所不同。
- 不同程式語言會以不同的方式實作執行緒，許多作業系統都有提供 API 來建立新的執行緒。Rust 標準函式庫使用的是 1:1 的執行緒實作模型，也就是每一個語言產生的執行緒就是一個作業系統的執行緒。有其他 crate 會實作其他種執行緒模型，能與 1:1 模型之間做取捨。

無懼並行

透過 spawn 建立新的執行緒

- 要建立一個新的執行緒，呼叫函式 thread::spawn 並傳入一個閉包，其包含想在新執行緒執行的程式碼。底下範例會在主執行緒印出一些文字，並在新執行緒印出其他文字：

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("數字 {} 出現在產生的執行緒中！", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("數字 {} 出現在主執行緒中！", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

注意到當 Rust 程式的主執行緒完成的話，所有執行緒也會被停止，無論它有沒有完成任務。此程式的輸出結果每次可能都會有點不相同，但它會類似以下這樣：

```
數字 1 出現在主執行緒中！
數字 1 出現在產生的執行緒中！
數字 2 出現在主執行緒中！
數字 2 出現在產生的執行緒中！
數字 3 出現在主執行緒中！
數字 3 出現在產生的執行緒中！
數字 4 出現在主執行緒中！
數字 4 出現在產生的執行緒中！
數字 5 出現在產生的執行緒中！
```

無懼並行

```rust
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("數字 {} 出現在產生的執行緒中！", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("數字 {} 出現在主執行緒中！", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

透過 spawn 建立新的執行緒

- thread::sleep 的呼叫強制執行緒短時間內停止運作，讓不同的執行緒可以執行。執行緒可能會輪流執行，但並不保證絕對如此，這會依據作業系統如何安排執行緒而有所不同。在這一輪中，主執行緒會先顯示，就算程式中是先寫新執行緒的 println! 陳述式。而且雖然是寫說新執行緒印出 i 一直到 9，但它在主執行緒結束前只印到 5。
- 如果當執行此程式時只看到主執行緒的結果，或者沒有看到任何交錯的話，可以嘗試增加數字範圍來增加作業系統切換執行緒的機會。

無懼並行

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("數字 {} 出現在產生的執行緒中！", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("數字 {} 出現在主執行緒中！", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

使用 join 等待所有執行緒完成

- 之前範例的程式碼在**主執行緒結束時不只會在大多數的時候提早結束新產生的執行緒**，還不能保證執行緒運行的順序，甚至無法保證產生的執行緒真的會執行！
- 透過儲存 thread::spawn 回傳的數值為變數，可以修正產生的執行緒完全沒有執行或沒有執行完成的問題。thread::spawn 的回傳型別為 JoinHandle。JoinHandle 是個有所有權的數值，當對它呼叫 join 方法時，它就會等待它的執行緒完成。
- 右上範例顯示了如何使用在之前範例中執行緒的 JoinHandle 並呼叫 join 來確保產生的執行緒會在 main 離開之前完成：

無懼並行

使用 join 等待所有執行緒完成

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("數字 {} 出現在產生的執行緒中！", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("數字 {} 出現在主執行緒中！", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

- 對其呼叫 join 會阻擋當前正在執行的執行緒中直到 JoinHandle 的執行緒結束為止。阻擋(Blocking)一條執行緒代表該執行緒不會繼續運作或離開。因為在主執行緒的 for 迴圈之後加上了 join 的呼叫，右上範例應該會產生類似以下的輸出：

```
數字 1 出現在主執行緒中！
數字 2 出現在主執行緒中！
數字 1 出現在產生的執行緒中！
數字 3 出現在主執行緒中！
數字 2 出現在產生的執行緒中！
數字 4 出現在主執行緒中！
數字 3 出現在產生的執行緒中！
數字 4 出現在產生的執行緒中！
數字 5 出現在產生的執行緒中！
數字 6 出現在產生的執行緒中！
數字 7 出現在產生的執行緒中！
數字 8 出現在產生的執行緒中！
數字 9 出現在產生的執行緒中！
```

兩條執行緒會互相交錯，但是主執行緒這次會因為 handle.join() 而等待，直到產生的執行緒完成前都不會結束。

無懼並行

使用 join 等待所有執行緒完成

- 那如果如以下這樣將 handle.join() 移到 main 中的 for 迴圈前會發生什麼事呢：

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("數字 {} 出現在產生的執行緒中！", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("數字 {} 出現在主執行緒中！", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

```
數字 1 出現在產生的執行緒中！
數字 2 出現在產生的執行緒中！
數字 3 出現在產生的執行緒中！
數字 4 出現在產生的執行緒中！
數字 5 出現在產生的執行緒中！
數字 6 出現在產生的執行緒中！
數字 7 出現在產生的執行緒中！
數字 8 出現在產生的執行緒中！
數字 9 出現在產生的執行緒中！
數字 1 出現在主執行緒中！
數字 2 出現在主執行緒中！
數字 3 出現在主執行緒中！
數字 4 出現在主執行緒中！
```

- 主執行緒會等待產生的執行緒完成才會執行它的 for 迴圈，所以輸出結果就不會彼此交錯，如右上所示：
- 像這樣將 join 呼叫置於何處的小細節，會影響執行緒會不會同時運行。

無懼並行

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("這是個向量：{:?}", v);
    });

    handle.join().unwrap();
}
```

透過執行緒使用 move 閉包

- 通常會使用 thread::spawn 時都會搭配有 move 關鍵字的閉包，因為該閉包能獲取周圍環境的數值，轉移那些數值的所有權到另一個執行緒中。在之前的「獲取參考或移動所有權」段落討論過閉包如何運用 move。現在會來專注在 move 與 thread::spawn 之間如何互動。
- 在之前提到可以在閉包參數列表前使用 move 關鍵字來強制閉包取得其從環境獲取數值的所有權。此技巧在建立新的執行緒特別有用，可以從一個執行緒轉移數值所有權到另一個執行緒。
- 注意到之前範例中傳入 thread::spawn 的閉包沒有任何引數，在產生的執行緒程式碼內沒有使用主執行緒的任何資料。要在產生的執行緒中使用主執行緒的資料的話，產生的執行緒閉包必須獲取它所需的資料。右上範例嘗試在主執行緒建立一個向量並在產生的執行緒使用它。不過這目前無法執行，會在稍後知道原因。

無懼並行

透過執行緒使用 move 閉包

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("這是個向量：{:?}", v);
    });

    handle.join().unwrap();
}
```

- 閉包想使用 v，所以它得獲取 v 並使其成為閉包環境的一部分。因為 thread::spawn 會在新的執行緒執行此閉包，要能在新的執行緒內存取 v。但當編譯此範例時，會得到以下錯誤：

```
$ cargo run
   Compiling threads v0.1.0 (file:///projects/threads)
error[E0373]: closure may outlive the current function, but it borrows `v`, which is owned by the
current function
 --> src/main.rs:6:32
  |
6 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `v`
7 |         println!("這是個向量：{:?}", v);
  |                                    - `v` is borrowed here
  |
note: function requires argument type to outlive `'static`
 --> src/main.rs:6:18
  |
6 |       let handle = thread::spawn(|| {
  |  __________________^
7 | |         println!("這是個向量：{:?}", v);
8 | |     });
  | |_____^
help: to force the closure to take ownership of `v` (and any other referenced variables), use the
`move` keyword
  |
6 |     let handle = thread::spawn(move || {
  |                                ++++

For more information about this error, try `rustc --explain E0373`.
error: could not compile `threads` due to previous error
```

Rust 會推斷如何獲取 v 而且因為 println! 只需要 v 的參考，閉包得借用 v。不過這會有個問題，Rust 無法知道產生的執行緒會執行多久，所以它無法確定 v 的參考是不是永遠有效。

無懼並行

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("這是個向量：{:?}", v);
    });

    drop(v); // 喔不！

    handle.join().unwrap();
}
```

透過執行緒使用 move 閉包

- 右上範例提供了一個情境讓 v 很有可能不再有效：
- 如果 Rust 允許執行此程式碼，產生的執行緒是有可能會置於背景而沒有馬上執行。產生的執行緒內部有 v 的參考，但主執行緒會立即釋放 v，使用在之前討論過的 drop 函式。然後當產生的執行緒開始執行時，v 就不再有效了，所以它的參考也是無效的了。
- 要修正右上範例的編譯錯誤，可以使用錯誤訊息的建議：

```
help: to force the closure to take ownership of `v` (and any other referenced variables), use the
`move` keyword
  |
6 |     let handle = thread::spawn(move || {
  |                                ++++
```

無懼並行

透過執行緒使用 move 閉包

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("這是個向量：{:?}", v);
    });

    handle.join().unwrap();
}
```

- 透過在閉包前面加上 move 關鍵字，強制讓閉包取得它所要使用數值的所有權，而非任由 Rust 去推斷它是否該借用數值。右上範例修改了之前範例並能夠如期編譯與執行：
- 可能會想嘗試用之前範例做的事來修正程式碼，使用 move 閉包的同時在主執行緒呼叫 drop。但這樣修正沒有用，之前範例想做的事情會因為不同原因而不被允許。如果對閉包加上了 move，將會把 v 移入閉包環境，而在主執行緒將無法再對它呼叫 drop 了。會得到另一個編譯錯誤：

```
$ cargo run
   Compiling threads v0.1.0 (file:///projects/threads)
error[E0382]: use of moved value: `v`
  --> src/main.rs:10:10
   |
4  |     let v = vec![1, 2, 3];
   |         - move occurs because `v` has type `Vec<i32>`, which does not implement the `Copy` trait
5  |
6  |     let handle = thread::spawn(move || {
   |                                ------- value moved into closure here
7  |         println!("這是個向量：{:?}", v);
   |                                    - variable moved due to use in closure
...
10 |     drop(v); // 喔不！
   |          ^ value used here after move

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

無懼並行

透過執行緒使用 move 閉包

```
$ cargo run
   Compiling threads v0.1.0 (file:///projects/threads)
error[E0382]: use of moved value: `v`
  --> src/main.rs:10:10
   |
4  |     let v = vec![1, 2, 3];
   |         - move occurs because `v` has type `Vec<i32>`, which does not implement the `Copy` trait
5  |
6  |     let handle = thread::spawn(move || {
   |                                ------- value moved into closure here
7  |         println!("這是個向量：{:?}", v);
   |                                    - variable moved due to use in closure
...
10 |     drop(v); // 喔不！
   |          ^ value used here after move

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

- Rust 的所有權規則再次拯救了！在之前範例會得到錯誤是因為 Rust 是保守的，所以只會為執行緒借用 v，這代表主執行緒理論上可能會使產生的執行緒的參考無效化。
- 透過告訴 Rust 將 v 的所有權移入產生的執行緒中，向 Rust 保證不會在主執行緒用到 v。如果用相同方式修改範例的話，當嘗試在主執行緒使用 v 的話，就違反了所有權規則。
- move 關鍵字會覆蓋 Rust 保守的預設借用行為，且也不允許違反所有權規則。有了對執行緒與執行緒 API 的基本瞭解，看看可以透過執行緒做些什麼。

無懼並行

使用訊息傳遞在執行緒間傳送資料

- 有一種確保安全並行且漸漸流行起來的方式是訊息傳遞(message passing)，執行緒透過傳遞包含資料的訊息給彼此來溝通。此理念源自於 Go 語言技術文件中的口號：「別透過共享記憶體來溝通，而是透過溝通來共享記憶體。」
- 對於訊息傳遞的並行，Rust 的標準函式庫有提供通道(channel)的實作。通道是一種程式設計的概念，會把資料從一個執行緒傳送到另一個。可以把程式設計的通道想像成水流的通道，像是河流或小溪。如果將橡皮小鴨或船隻放入河流中，它會順流而下到下游。
- 一個通道會包含兩個部分：發送者(transmitter)與接收者(receiver)。發送者正是會放置橡皮小鴨到河流中的上游，而接收者則是橡皮小鴨最後漂流到的下游。程式碼中的一部分會呼叫發送者的方法來傳送想要傳遞的資料，然後另一部分的程式碼會檢查接收者收到的訊息。當發送者或接收者有一方被釋放掉時，該通道就會被關閉。

無懼並行

使用訊息傳遞在執行緒間傳送資料

- 在此將寫一支程式，它會在一個執行緒中產生數值，傳送給通道，然後另一個執行緒會接收到數值並印出來。會使用通道在執行緒間傳送簡單的數值來作為這個功能的解說。熟悉此技巧後，可以使用通道讓執行緒間可以互相溝通。像是實作個聊天系統，或是一個利用數個執行緒進行運算，然後將結果傳入一個執行緒統整結果的分散式系統。
- 首先在底下範例會建立個通道但還不會做任何事。注意這樣不會編譯通過因為 Rust 無法知道想對通道傳入的數值型別為何：

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
}
```

無懼並行

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
}
```

使用訊息傳遞在執行緒間傳送資料

- 使用 mpsc::channel 函式來建立新的通道，mpsc 指的是多重生產者、唯一消費者(multiple producer, single consumer)。簡單來說，Rust 標準函式庫實作通道的方式讓通道可以有**多個**發送端來產生數值，不過只有**一個**接收端能消耗這些數值。
  - 想像有數個溪流匯聚成一條大河流，任何溪流傳送的任何東西最終都會流向河流的下游。
  - 會先從單一生產者開始，等這個範例能夠執行後再來增加數個生產者。
- mpsc::channel 函式會回傳一個元組，第一個元素是發送者然後第二個元素是接收者。
- tx 與 rx 通常分別作為發送者(transmitter)與接收者(receiver)的縮寫，所以以此作為變數名稱。let 陳述式使用到了能解構元組的模式，會在之後討論 let 陳述式的模式與解構方式。
- 用這樣的方式使用 let 能輕鬆取出 mpsc::channel 回傳的元組每個部分。

無懼並行

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("嗨");
        tx.send(val).unwrap();
    });
}
```

使用訊息傳遞在執行緒間傳送資料

- 將發送端移進一個新產生的執行緒並讓它傳送一條字串，這樣產生的執行緒就可以與主執行緒溝通了，如右上所示。這就像是在河流上游放了一隻橡皮小鴨，或是從一條執行緒傳送一條聊天訊息給別條執行緒一樣：
- 再次使用 thread::spawn 來建立新的執行緒並使用 move 將 tx 移入閉包，讓產生的執行緒擁有 tx。產生的執行緒必須要擁有發送者才能夠傳送訊息至通道。發送端有個 send 方法可以接受想傳遞的數值。
- send 方法會回傳 Result<T, E> 型別，所以如果接收端已經被釋放因而沒有任何地方可以傳遞數值的話，傳送的動作就會回傳錯誤。在此例中，呼叫 unwrap 所以有錯誤時就會直接恐慌。但在實際的應用程式中，會更妥善地處理它，可以回顧之前章節來複習如何適當地處理錯誤。

無懼並行

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("嗨");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("取得：{}", received);
}
```

使用訊息傳遞在執行緒間傳送資料

- 在右上範例會在主執行緒中從接收者取得數值。這就像在河流下游取回順流而下的橡皮小鴨，或是像取得一條聊天訊息一樣：
- 接收者有兩個實用的方法：recv 與 try_recv。使用 **recv** 作為接收(receive)的縮寫，這會**阻擋主執行緒**的運行並等待直到通道有訊息傳入。一旦有數值傳遞，recv 會就以此回傳 Result<T, E>。當發送者關閉時，recv 會回傳錯誤來通知不會再有任何數值出現了。
- **try_recv方法則不會阻擋**，而是會立即回傳Result<T, E>。如果有數值的話，就會是存有訊息的Ok數值，如果尚未有任何數值的話，就會是 Err 數值。try_recv 適用於如果此執行緒在等待訊息的同時有其他事要做的情形。可以寫個迴圈來時不時呼叫 try_recv，當有數值時處理訊息，不然的話就先做點其他事直到再次檢查為止。
- 出於方便考量在此例使用 recv，主執行緒除了等待訊息以外沒有其他事好做，所以阻擋主執行緒是合理的。當執行右上範例的程式碼，會看到主執行緒印出的數值：

```
取得：嗨
```

無懼並行

通道與所有權轉移

- 所有權規則在訊息傳遞中扮演了重要的角色，因為它們可以幫助寫出安全的並行程式碼。在 Rust 程式中考慮所有權的其中一項好處就是能在並行程式設計避免錯誤發生。
- 做個實驗來看通道與所有權如何一起合作來避免問題發生，會在 val 數值傳送給通道之後嘗試使用其值。請嘗試編譯右下範例的程式碼並看看為何此程式碼**不被允許**：

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("嗨");
        tx.send(val).unwrap();
        println!("val 為 {}", val);
    });

    let received = rx.recv().unwrap();
    println!("取得：{}", received);
}
```

無懼並行

通道與所有權轉移

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("嗨");
        tx.send(val).unwrap();
        println!("val 為 {}", val);
    });

    let received = rx.recv().unwrap();
    println!("取得：{}", received);
}
```

- 在這裡透過 tx.send 將 val 傳入通道之後嘗試印出其值。允許這麼做的話會是個壞主意，一旦數值被傳至其他執行緒，**該執行緒就可以在嘗試再次使用該值之前修改或釋放其值**。
- 其他執行緒的修改有機會因為不一致或不存在的資料而導致錯誤或意料之外的結果。不過如果試著編譯右上範例的程式碼的話，Rust 會給一個錯誤：

```
$ cargo run
   Compiling message-passing v0.1.0 (file:///projects/message-passing)
error[E0382]: borrow of moved value: `val`
  --> src/main.rs:10:31
   |
8  |         let val = String::from("嗨");
   |             --- move occurs because `val` has type `String`, which does not implement the `Copy`
   trait
9  |         tx.send(val).unwrap();
   |                 --- value moved here
10 |         println!("val 為 {}", val);
   |                               ^^^ value borrowed here after move
   |
   = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion
   of the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)

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

並行錯誤產生了一個編譯時錯誤。send 函式會取走其參數的所有權，並當數值移動時，接收端會再取得其所有權。這能阻止在傳送數值過後不小心再次使用其值，所有權系統會檢查一切是否符合規則。

無懼並行

傳送多重數值並觀察接收者等待

- 之前範例程式碼可以編譯通過並執行，但它並沒有清楚表達兩個不同的執行緒正透過通道彼此溝通。在底下範例中，做了些修改來證明之前範例有正確執行，產生的執行緒現在會傳送數個訊息，並在每個訊息間暫停個一秒鐘：
- 這次產生的執行緒有個字串向量，希望能傳送它們到主執行緒中。走訪它們，單獨傳送每個值，然後透過 Duration 數值呼叫 thread::sleep 來暫停一秒。
- 在主執行緒中，不再顯式呼叫 recv 函式，改將 rx 作為疊代器使用。對每個接收到的數值，印出它。
- **當通道關閉時，疊代器就會結束。**

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("執行緒"),
            String::from("傳來"),
            String::from("的"),
            String::from("嗨"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("取得：{}", received);
    }
}
```

無懼並行

傳送多重數值並觀察接收者等待

- 當執行範例的程式碼，應該會看到以下輸出，每一行會間隔一秒鐘：

```
取得：執行緒
取得：傳來
取得：的
取得：嗨
```

- 因為在主執行緒中的 for 迴圈內沒有任何會暫停或延遲的程式碼，所以可以看出主執行緒是在等待產生的執行緒傳送的數值。

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let vals = vec![
            String::from("執行緒"),
            String::from("傳來"),
            String::from("的"),
            String::from("嗨"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("取得：{}", received);
    }
}
```

無懼並行

透過clone發送者來建立多重生產者

- 之前提過 mpsc 是多重生產者、唯一消費者(multiple producer, single consumer)的縮寫。
- 來使用 mpsc 並擴展之前範例的程式碼來建立數個執行緒，它們都將傳遞數值給同個接收者。為此可以clone發送者，如右邊範例所示：
- 這次在建立第一個產生的執行緒前，會對發送者呼叫 clone。這能給一個新的發送者，可以移入第一個產生的執行緒。
- 接著將原本的通道發送端移入第二個產生的執行緒中。這樣就有了兩條執行緒，每條都能傳送不同的訊息給接收者。

```
// --省略--

let (tx, rx) = mpsc::channel();

let tx1 = tx.clone();
thread::spawn(move || {
    let vals = vec![
        String::from("執行緒"),
        String::from("傳來"),
        String::from("的"),
        String::from("嗨"),
    ];

    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

thread::spawn(move || {
    let vals = vec![
        String::from("更多"),
        String::from("給你"),
        String::from("的"),
        String::from("訊息"),
    ];

    for val in vals {
        tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

for received in rx {
    println!("取得：{}", received);
}

// --省略--
```

無懼並行

透過clone發送者來建立多重生產者

- 當執行程式碼時，輸出應該會類似以下結果

```
取得：執行緒
取得：更多
取得：傳來
取得：給你
取得：的
取得：的
取得：嗨
取得：訊息
```

- 依據系統可能會看到數值以不同順序排序。這正是並行程式設計既有趣卻又困難的地方。如果加上 thread::sleep 來實驗，並在不同執行緒給予不同數值的話，就會發現每一輪都會更不確定，每次都會產生不同的輸出結果。
- 現在已經看完通道如何運作，接著來看看並行的不同方法吧。

```rust
// --省略--

let (tx, rx) = mpsc::channel();

let tx1 = tx.clone();
thread::spawn(move || {
    let vals = vec![
        String::from("執行緒"),
        String::from("傳來"),
        String::from("的"),
        String::from("嗨"),
    ];

    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

thread::spawn(move || {
    let vals = vec![
        String::from("更多"),
        String::from("給你"),
        String::from("的"),
        String::from("訊息"),
    ];

    for val in vals {
        tx.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});

for received in rx {
    println!("取得：{}", received);
}

// --省略--
```

無懼並行

共享狀態並行

- 雖然之前說過 Go 語言技術文件中的口號：「別透過共享記憶體來溝通。」而訊息傳遞是個很好的並行處理方式，但它不是唯一的選項。另一種方式就是在**多重執行緒間共享資料**。透過共享記憶體來溝通會是什麼樣子呢？除此之外，為何訊息傳遞愛好者不喜歡這種共享記憶體的方式呢？
- 任何程式語言的**通道**某方面來說類似於**單一所有權**，因為一旦轉移數值給通道，就不該使用該數值。**共享記憶體**並行則像**多重所有權**，數個執行緒可以同時存取同個記憶體位置。
- 如同在之前章節所見到的，**智慧指標讓多重所有權成為可能**，但多重所有權會增加複雜度，因為會需要管理這些不同的擁有者。Rust 的型別系統與所有權規則大幅地協助了正確管理這些所有權。作為範例就來看看互斥鎖(mutexes)，這是共享記憶體中常見的並行原始元件之一。

無懼並行

使用互斥鎖在同時間只允許一條執行緒存取資料

- 互斥鎖(Mutex)是 mutual exclusion 的縮寫，顧名思義互斥鎖在任意時刻只允許一條執行緒可以存取一些資料。要取得互斥鎖中的資料，執行緒必須先透過獲取互斥鎖的鎖(lock)來表示它想要進行存取。鎖是互斥鎖其中一部分的資料結構，用來追蹤當前誰擁有資料的獨佔存取權。因此互斥鎖被描述為會透過鎖定系統守護(guarding)其所持有的資料。
- 互斥鎖以難以使用著名，因為必須記住兩個規則：
  1. 必須在使用資料前獲取鎖。
  2. 當用完互斥鎖守護的資料，必須解鎖資料，所以其他的執行緒才能獲取鎖。

無懼並行

使用互斥鎖在同時間只允許一條執行緒存取資料

- 要用真實世界來比喻互斥鎖的話，想像在會議中有個座談會只有一支麥克風。如果有講者想要發言時，請求或示意他們想要使用麥克風。當取得麥克風時，
- 想講多久都沒問題，直到將麥克風遞給下個要求發言的講者。如果講者講完後忘記將麥克風遞給其他人的話，就沒有人有辦法發言。如果麥克風的分享出狀況的話，座談會就無法如期進行！
- 互斥鎖的管理要正確處理是極為困難的，這也是為何這麼多人傾向於使用通道。然而有了 Rust 的型別系統與所有權規則，就不會在鎖定與解鎖之間出錯了。

無懼並行

```rust
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {:?}", m);
}
```

Mutex<T> 的 API

- 作為使用互斥鎖的範例，先在單執行緒使用互斥鎖，如右上範例所示：
- 就像許多型別一樣，使用關聯函式 new 建立 Mutex<T>。要取得互斥鎖內的資料，使用 lock 方法來獲取鎖。此呼叫會阻擋當前執行緒做任何事，直到輪到它取得鎖。
- 如果其他持有鎖的執行緒恐慌的話 lock 的呼叫就會失敗。在這樣的情況下，就沒有任何人可以獲得鎖，因此當遇到這種情況時，選擇 unwrap 並讓此執行緒恐慌。
- 在獲取鎖之後，在此例可以將回傳的數值取作 num，作為內部資料的可變參考。型別系統能確保在使用數值 m 之前有獲取鎖，**Mutex<i32> 並不是 i32，所以必須取得鎖才能使用 i32 數值**。不可能會忘記這麼做，不然型別系統不給存取內部的 i32。

無懼並行

```rust
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {:?}", m);
}
```

Mutex<T> 的 API

- 如同所想像的，Mutex<T> 就是個智慧指標。更精確的來說，lock 的呼叫會回傳一個智慧指標叫做 MutexGuard，這是從 LockResult 呼叫 unwrap 取得的型別。
- MutexGuard 智慧指標有實作 **Deref** 特徵來指向內部資料。此智慧指標也有 **Drop** 的實作，這會在 MutexGuard 離開作用域時自動釋放鎖，也就是在內部作用域結尾就會執行此動作。這樣一來，就不會忘記釋放鎖，怕互斥鎖會阻擋其他執行緒，因為鎖會自動被釋放。
- 在釋放鎖之後，就能印出互斥鎖的數值並觀察到能夠變更內部的 i32 為 6。

無懼並行

```rust
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("結果：{}", *counter.lock().unwrap());
}
```

在數個執行緒間共享 Mutex<T>

- 現在來透過 Mutex<T> 來在數個執行緒間分享數值。
- 建立 10 個執行緒並讓它們都會對一個計數增加 1，讓計數能從 0 加到 10。作為下個例子的右上範例會出現一個編譯錯誤，會用此錯誤瞭解如何使用 Mutex<T> 以及 Rust 如何協助來正確使用它：
- 建立個變數 counter 並在 Mutex<T> 內存有 i32，就像在之前範例所做的一樣。接著透過指定的範圍建立10個執行緒。使用 thread::spawn 讓所有的執行緒都有相同的閉包，此閉包會將計數移入執行緒、呼叫lock以獲取Mutex<T>的鎖，然後將互斥鎖內的數值加 1。
- 當有執行緒執行完它的閉包時，num會離開作用域並釋放鎖，讓其他的執行緒可以獲取它。

無懼並行

在數個執行緒間共享 Mutex<T>

```rust
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("結果：{}", *counter.lock().unwrap());
}
```

- 在主執行緒中，要收集所有的執行緒。然後如同在之前範例所做的，呼叫每個執行緒的 join 來確保所有執行緒都有完成。在這時候，主執行緒就能獲取鎖並印出此程式的結果。
- 曾暗示範例不會編譯過，來看看是為何吧：

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0382]: use of moved value: `counter`
  --> src/main.rs:9:36
   |
5  |     let counter = Mutex::new(0);
   |         ------- move occurs because `counter` has type `Mutex<i32>`, which does not implement the
   `Copy` trait
...
9  |         let handle = thread::spawn(move || {
   |                                    ^^^^^^^ value moved into closure here, in previous iteration
   of loop
10 |             let mut num = counter.lock().unwrap();
   |                           ------- use occurs due to use in closure

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

- 錯誤訊息表示 counter 數值在之前的迴圈循環中被移動了，所以 Rust 告訴無法將 counter 鎖的所有權移至數個執行緒中。用之前提到的多重所有權方法來修正此編譯錯誤吧。

無懼並行

多重執行緒中的多重所有權

- 在之前透過智慧指標 Rc<T> 來建立參考計數數值讓該資料可以擁有數個擁有者。
- 在此也做同樣的動作來看看會發生什麼事。會在底下範例將 Mutex<T> 封裝進 Rc<T> 並在將所有權移至執行緒前clone Rc<T>：

```
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("結果：{}", *counter.lock().unwrap());
}
```

無懼並行

多重執行緒中的多重所有權

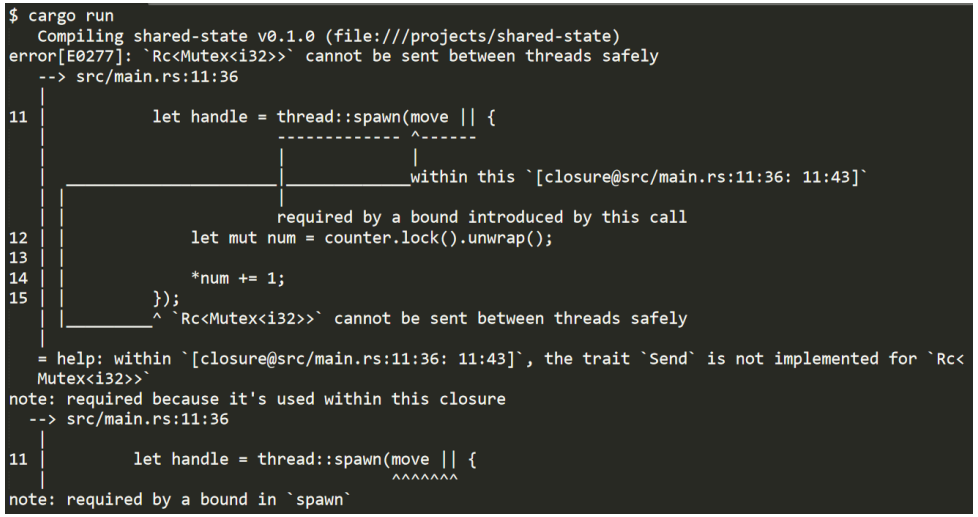
- 再編譯一次的話會得到... 不同的錯誤：

```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0277]: `Rc<Mutex<i32>>` cannot be sent between threads safely
   --> src/main.rs:11:36
    |
11  |           let handle = thread::spawn(move || {
    |                        ------------ ^------
    |                        |            |
    |  ______________________|____________within this `[closure@src/main.rs:11:36: 11:43]`
    | |                      |
    | |                      required by a bound introduced by this call
12  | |               let mut num = counter.lock().unwrap();
13  | |
14  | |               *num += 1;
15  | |           });
    | |___________^ `Rc<Mutex<i32>>` cannot be sent between threads safely
    |
    = help: within `[closure@src/main.rs:11:36: 11:43]`, the trait `Send` is not implemented for `Rc<
   Mutex<i32>>`
note: required because it's used within this closure
  --> src/main.rs:11:36
   |
11 |           let handle = thread::spawn(move || {
   |                                      ^^^^^^^
note: required by a bound in `spawn`
```

```rust
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("結果：{}", *counter.lock().unwrap());
}
```

無懼並行

多重執行緒中的多重所有權



```
$ cargo run
   Compiling shared-state v0.1.0 (file:///projects/shared-state)
error[E0277]: `Rc<Mutex<i32>>` cannot be sent between threads safely
   --> src/main.rs:11:36
    |
11  |         let handle = thread::spawn(move || {
    |                      ------------- ^------
    |                      |             |
    |  ____________________|_____________within this `[closure@src/main.rs:11:36: 11:43]`
    | |                    |
    | |                    required by a bound introduced by this call
12  | |             let mut num = counter.lock().unwrap();
13  | |
14  | |             *num += 1;
15  | |         });
    | |_________^ `Rc<Mutex<i32>>` cannot be sent between threads safely
    |
    = help: within `[closure@src/main.rs:11:36: 11:43]`, the trait `Send` is not implemented for `Rc<
   Mutex<i32>>`
note: required because it's used within this closure
  --> src/main.rs:11:36
   |
11 |         let handle = thread::spawn(move || {
   |                                    ^^^^^^^
note: required by a bound in `spawn`
```

- 需要注意到的部分：`Rc<Mutex<i32>>` cannot be sent between threads safely。
- 編譯器也告訴了原因：the trait `Send` is not implemented for `Rc<Mutex<i32>>` 。會在下一個段落討論 Send，這是其中一種**確保在執行緒中所使用的型別可以用於並行場合的特徵**。
- 不幸的是 Rc<T> 無法安全地跨執行緒分享。當 Rc<T> 管理參考計數時，它會在每個 clone 的呼叫增加計數，並在每個clone釋放時減少計數。
- 但是它沒有使用任何並行原始元件來確保計數的改變不會被其他執行緒中斷。這樣的計數可能會導致微妙的程式錯誤，像是記憶體泄漏或是在數值釋放時嘗試使用其值。需要一個型別和 Rc<T> 一模一樣，但是其參考計數在執行緒間是安全的。

無懼並行

原子參考計數 Arc<T>

- 幸運的是 **A**rc<T> 正是一個類似 Rc<T> 且能安全用在並行場合的型別。
- 字母 A 指的是原子性(atomic)代表這是個原子性參考的計數型別。原子型別是另一種不會在此討論的並行原始元件，可以查閱標準函式庫的 std::sync::atomic 技術文件以瞭解更多詳情。在此只需要知道原子型別和原始型別類似，但它們**可以安全在執行緒間共享**。
- 可能會好奇為何原始型別不是原子性的，以及為何標準函式庫的型別預設不使用 Arc<T> 來實作。原因是因為執行緒安全意味著效能開銷，會希望在真的需要時才買單。
- 如果只是在單一執行緒對數值做運算的話，程式碼就不必強制具有原子性的保障並能執行更快。

無懼並行

原子參考計數 Arc<T>

- 回到範例：Arc<T> 與 Rc<T> 具有相同的 API，所以透過更改 use 這行、new 的呼叫以及 clone 的呼叫來修正程式。底下範例的程式碼最終將能夠編譯並執行：
- 此程式碼會印出以下結果：`結果：10`

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("結果：{}", *counter.lock().unwrap());
}
```

無懼並行

原子參考計數 Arc<T>

- 從 0 數到了 10，雖然看起來不是很令人印象深刻，但這的確教會了很多有關 Mutex<T> 與執行緒安全的知識。也可以使用此程式結構來做更多複雜的運算，而不只是數數而已。
- **使用此策略，可以將運算拆成數個獨立部分，將它們分配給執行緒，然後使用 Mutex<T> 來讓每個執行緒更新該部分的結果。**
- 如果只是想做單純的數值運算，其實標準函式庫提供的 std::sync::atomic 模組會比 Mutex<T> 型別來得簡單。這些型別對原生型別提供安全、並行且原子性的順取。
- 在此例對原生型別使用 Mutex<T>，是因為想解釋 Mutex<T> 是如何運作的。

無懼並行

RefCell<T>/Rc<T> 與 Mutex<T>/Arc<T> 之間的相似度

- 可能已經注意到 counter 是不可變的，但卻可以取得數值其內部的可變參考，這代表 Mutex<T> 有提供內部可變性，就像 Cell 家族一樣。**在之前也以相同的方式使用 RefCell<T> 來改變 Rc<T> 內部的數值，而在此使用 Mutex<T> 改變 Arc<T> 內部的內容。**
- 另一個值得注意的細節是當使用 Mutex<T> 時，**Rust 無法避免所有種類的邏輯錯誤**。
- **回憶一下之前使用 Rc<T> 時會有可能產生參考循環的風險，兩個 Rc<T> 數值可能會彼此參考，造成記憶體泄漏。同樣地，Mutex<T> 有產生死結(deadlocks)的風險。**
- 這會發生在當有個動作需要鎖定兩個資源，而有兩個執行緒分別擁有其中一個鎖，導致它們永遠都在等待彼此。如果對此有興趣的話，歡迎嘗試建立一個有死結的 Rust 程式，然後研究看看任何語言中避免的互斥鎖死結的策略，並嘗試實作它們在 Rust 中。
- **標準函式庫中 Mutex<T> 與 MutexGuard 的 API 技術文件可以提供些實用資訊。**

無懼並行

可延展的並行與 Sync 及 Send 特徵

- 有一個有趣的點是 Rust 語言提供的並行功能並沒有很多。這邊討論到的並行功能幾乎都來自於標準函式庫，並不是語言本身。能處理並行的選項並不限於語言或標準函式庫，可以寫出自己的並行功能或使用其他人提供的。
- 然而，還有有兩個並行概念深植於語言中，那就是 std::marker 中的 Sync 與 Send 特徵。

無懼並行

透過 Send 來允許所有權能在執行緒間轉移

- Send 標記特徵(marker traits)指定有實作 Send 特徵的型別才能將其數值的所有權在執行緒間轉移。幾乎所有的 Rust 型別都有 Send，但有些例外。
- 這包含 Rc<T>，此型別沒有 Send 是因為如果clone了 Rc<T> 數值並嘗試轉移clone的所有權到其他執行緒，會有兩條執行緒可能同時更新參考計數。基於此原因，Rc<T> 是用於當不想要付出執行緒安全效能開銷時而在單一執行緒使用的情況。
- 因此 Rust 的型別系統與特徵界限確保無法意外不安全地傳送 Rc<T> 數值到其他執行緒。當嘗試之前範例時，就會得到錯誤 the trait Send is not implemented for Rc<Mutex<i32>>。**當切換成有實作 Send 的 Arc<T> 的話，程式碼就能編譯通過。**
- 任何由具有 Send 的型別所組成的型別也都會自動標記為 Send。幾乎所有原始型別都是 Send，除了將在之後提及的**裸指標(raw pointers)**。

無懼並行

透過 Sync 來允許多重執行緒存取

- Sync 標記特徵指定有實作 Sync 的型別都能安全從多個執行緒來參考。換句話說，對於任何型別 T，如果 &T(對 T 的不可變參考)有 Send 的話，T 就是 Sync 的，這代表參考可以安全地傳給其他執行緒。與 Send 類似，原始型別都是 Sync，所以由具有 Sync 的型別所組成的型別也都有 Sync。
- 智慧指標 Rc<T> 沒有 Sync 的原因和沒有 Send 的原因一樣。RefCell<T> 型別與其 Cell<T> 也都沒有 Sync。
- **RefCell<T> 在執行時的借用檢查實作沒有執行緒安全。智慧指標 Mutex<T> 才有 Sync 並能像在「在數個執行緒間共享 Mutex<T>」段落看到的那樣用來在多個執行緒間分享存取。**