控制流程(Control Flow)-

if/else

控制流程(Control Flow) - if/else

陳述式(statements)與表達式(expression)

- Rust 是基於表達式(expression-based)的語言，知道這樣的區別是很重要，其他語言通常沒有這樣的區別，
  - **陳述式(Statements)**：執行指定的一系列操作，且**不回傳任何數值，不會回傳結果**

```
fn main() {
    let a = 2;
}
```

控制流程(Control Flow) - if/else

陳述式(statements)與表達式(expression)

- Rust 是基於表達式(expression-based)的語言，知道這樣的區別是很重要，其他語言通常沒有這樣的區別，
  - **表達式(Expressions)**：則是計算並產生數值，經常透過一些符號結合上下語句並**運算及回傳結果**
    - 表達式則會運算出一個數值，並組合成大部分所寫的 Rust 程式。先想想看一個數學運算比如 5 + 6，這就是個會算出 11 的表達式。表達式可以是陳述式的一部分：let y = 6; 的 6 其實就是個算出 6 的表達式。呼叫函式也可以是表達式、呼叫巨集也是表達式、**用 {} 產生的作用域也是表達式**。

```
fn main() {
    let a = 8;

    let b = {
        let a = 4;
        a + 1
    };

    println!("a 的數值為：{a}");
}
```

```
> cargo run
   Compiling my-project v0.1.0 (/home/runner/test-6)
warning: unused variable: `b`
 --> src/main.rs:4:9
  |
4 |     let b = {
  |         ^ help: if this is intentional, prefix it with an underscore: `_b
`
  |
  = note: `#[warn(unused_variables)]` on by default

warning: `my-project` (bin "my-project") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 1.82s
     Running `target/debug/my-project`
a 的數值為：8
```

    - 就是一個會回傳 5 的區塊，此值再用 let 陳述式賦值給 b。**請注意到 a + 1 這行沒有加上分號**，它和目前看到的寫法有點不同，**因為表達式結尾不會加上分號。如果在此表達式加上分號的話，它就不會回傳數值。**

控制流程(Control Flow) - if/else

在大多數程式語言中，能夠決定依據某項條件是否為 true 來執行些程式碼，以及依據某項條件是否為 true 來重複執行些程式碼是非常基本的組成元件。在 Rust 程式碼中能控制執行流程的常見方法有 if 表達式以及迴圈。

依據某些條件是否為true，來決定是否執行：

- if
- else if
- else

控制流程(Control Flow) - if/else

if 表達式

- if 能依照條件判斷對程式碼產生分支。基本上提供一個條件然後就像是在說：「如果此條件符合的話，就執行這個程式碼區塊；如果沒有的話，就不要執行這段程式碼。」

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("條件為真");
    } else {
        println!("條件為否");
    }
}
```

所有的 if 表達式都由 if 關鍵字開始再加上一個條件。在此例中的條件是判斷變數number是否小於5。條件符合時所要執行的程式碼區塊被放在條件之後的大括號裡。與 if 表達式條件相關的程式碼段落有時也被稱為分支(arms)。

另外還可以**選擇性地**加上 else 表達式，讓條件不符時可以去執行另外一段程式碼。如果沒有提供 else 表達式且條件為否的話，程式會直接略過 if 的程式碼區塊，接著執行後續的程式碼。

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
條件為真
```

控制流程(Control Flow) - if/else

if 表達式

- 變更 number 的值使條件變成 false，再來看看會發生什麼事：

```
let number = 7;
```

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
條件為否
```

- 還有一點值得注意的是程式碼的**條件判斷必須是 bool**。如果條件不是 bool 的話，就會遇到錯誤。

  比方說，試試以下程式碼：

```
fn main() {
    let number = 3;

    if number {
        println!("數字為三");
    }
}
```

- 這次 if 條件計算出數值 3，然後 Rust 丟出錯誤：

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |     if number {
  |        ^^^^^^ expected `bool`, found integer

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

控制流程(Control Flow) - if/else

if 表達式

- 錯誤訊息顯示， Rust 預期收到 bool 但是卻拿到整數。這和 Ruby 和 JavaScript 就不同，**Rust 不會自動將非布林值型別轉換成布林值**。永遠必須顯式提供布林值給 if 作為它的條件判斷。
- 舉例來說，如果希望 if 只會在數值不為 0 才執行，可以將 if 表達式改成以下範例：

```
fn main() {
    let number = 3;

    if number != 0 {
        println!("數字不為零");
    }
}
```

- 執行此程式碼就會印出「數字不為零」。

控制流程(Control Flow) - if/else

使用 else if 處理多重條件

- 想要實現多重條件的話，可以將 if 和 else 組合成 else if 表達式。舉例來說：

```rust
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("數字可以被 4 整除");
    } else if number % 3 == 0 {
        println!("數字可以被 3 整除");
    } else if number % 2 == 0 {
        println!("數字可以被 2 整除");
    } else {
        println!("數字無法被 4、3、2 整除");
    }
}
```

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/branches`
數字可以被 3 整除
```

當此程式執行時，會依序檢查每一個 if 表達式，並執行第一個條件為 true 的程式碼段落。注意到雖然 6 的確可以除以 2，但沒有看到數字可以被 2 整除，也沒有看到來自 else 那段的數字無法被 4、3、2 整除。這是因為 Rust 只會執行第一個條件為 true 的區塊，而當它遇到時它就不會再檢查其他條件。

- 過多的else if會造成混亂，**未來會提到功能強大的 Rust 條件判斷結構 - match**

控制流程(Control Flow) - if/else

在 let 陳述式中使用 if

- 因為 if 是表達式，所以可以之前這樣放在 let 陳述式的右邊，將結果賦值給變數：

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("數字結果為：{number}");
}
```

- 變數 number 會得到 if 表達式運算出的數值。執行此程式看看會發生什麼事：

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/branches`
數字結果為：5
```

控制流程(Control Flow) - if/else

在 let 陳述式中使用 if

- 應該還記得程式碼區塊也可以是表達式且會回傳最後一行的數值，而且數字本身也是表達式。在此例中，if 表達式的值取決於哪段程式碼被執行。這代表可能成為最終結果的**每一個 if 分支必須要是相同型別**。在範例中，各分支的型別都是 i32。如果型別不一致的話，如以下所示：

```
fn main() {
    let condition = true;

    let number = if condition { 5 } else { "六" };

    println!("數字結果為：{number}");
}
```

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: `if` and `else` have incompatible types
 --> src/main.rs:4:44
  |
4 |     let number = if condition { 5 } else { "六" };
  |                                 -          ^^^^^ expected integer, found `&str`
  |                                 |
  |                                 expected because of this

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

- 當嘗試編譯程式碼時，得到錯誤。if 和 else 分支的型別並不一致，而且 Rust 還確切指出程式出錯的地方在哪：

控制流程(Control Flow) - if/else

```rust
fn main() {
    let condition = true;

    let number = if condition { 5 } else { "六" };

    println!("數字結果為：{number}");
}
```

在 let 陳述式中使用 if

- if 段落的表達式運算出整數，但 else 的區塊卻運算出字串。**這樣行不通的原因是變數只能有一個型別**。**Rust 必須在編譯期間確切知道變數 number 的型別**，這樣才能驗證它的型別在任何有使用到 number 的地方都是有效的。要是 number 只能在執行時知道的話，Rust 就沒辦法這樣做了。如果編譯器必須追蹤所有變數多種可能存在的型別，那就會變得非常複雜並無法為程式碼提供足夠的保障。

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: `if` and `else` have incompatible types
 --> src/main.rs:4:44
  |
4 |     let number = if condition { 5 } else { "六" };
  |                                -          ^^^^^ expected integer, found `&str`
  |                                |
  |                                expected because of this

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

控制流程(Control Flow)-

Loop & While & For

控制流程(Control Flow) - Loop & While Loop

使用迴圈重複執行

- 重複執行同一段程式碼區塊時常是很有用的。針對這樣的任務，Rust 提供了多種產生迴圈(loops)的方式。一個迴圈會執行一段程式碼區塊，然後在結束時馬上回到區塊起始位置繼續執行。Rust 提供三種迴圈：**loop**、**while** 和 **for**。

使用 loop 重複執行程式碼

- loop 關鍵字告訴 Rust 去反覆不停地執行一段程式碼直到親自告訴它要停下來：

```
fn main() {
    loop {
        println!("再一次！");
    }
}
```

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.29s
     Running `target/debug/loops`
再一次！
再一次！
再一次！
再一次！
^C再一次！
```

控制流程(Control Flow) - Loop & While Loop

```rust
fn main() {
    loop {
        println!("再一次！");
    }
}
```

使用 loop 重複執行程式碼

- 當執行此程式時，會看到 再一次！ 一直不停地重複顯示出來，直到手動停下程式為止。大多數的終端機都支援 ctrl-c 這個快捷鍵來中斷一支卡在無限迴圈的程式，可以試試看：

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.29s
     Running `target/debug/loops`
再一次！
再一次！
再一次！
再一次！
^C再一次！
```

```rust
fn main() {
    let mut n = 1;
    loop {
        println!("{:?}", n);
        if n == 4 {
            break;
        }
        n = n + 1;
    }
}
```

```
/*
Output
1
2
3
4
*/
```

- ^C 這個符號表示按下了 ctrl-c。按照程式收到中斷訊號的時間點，可能不會看到再一次！ 出現在 ^C 之後。
- Rust 有提供一個從程式中打破迴圈的方法。可以在迴圈內加上 break 關鍵字告訴程式何時停止執行迴圈。
- **Continue是會告訴程式跳過(略過)這次疊代中剩餘的程式碼，然後進行下一次疊代。**

控制流程(Control Flow) - Loop & While Loop

從迴圈回傳數值

- 其中一種使用 loop 的用途是重試某些覺得會失敗的動作，像是檢查一個執行緒是否已經完成其任務。**這樣可能就會想傳遞任務結果給之後的程式碼。要做到這樣的事，可以在要用來停下迴圈的 break 表達式內加上一個想回傳數值，該值就會被停止的迴圈回傳**，如以下所示：

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("結果為：{result}");
}
```

在迴圈之前，宣告了一個變數 counter 並初始化為 0，然後宣告了另一個變數 result 來取得迴圈回傳的值。在迴圈每一次的疊代中，將變數 counter 加上 1 並檢查它是否等於 10。如果是的話就用 break 關鍵字回傳 counter * 2。在迴圈結束後，用分號才結束這個賦值給 result 的陳述式。最後印出 result，而結果為 20。

控制流程(Control Flow) - Loop & While Loop

用迴圈標籤辨別多重迴圈

- 如果有迴圈在迴圈之內的話，break 和 continue 會用在該位置最內層的迴圈中。可以選擇在迴圈使用迴圈標籤(loop label)，然後使用 break 和 continue 加上那些迴圈標籤定義的關鍵字，而不是作用在最內層迴圈而已。以下是使用雙層巢狀迴圈的範例：

```
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

外層迴圈有個 'counting_up 的標籤，而且其會從 0 數到 2。而內層沒有迴圈標籤的迴圈則會從 10 數到 9。第一個 break 沒有指定任何標籤只會離開內層迴圈。而陳述式 break 'counting_up; 則會離開外層迴圈。此程式碼會印出：

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.58s
     Running `target/debug/loops`
count = 0
remaining = 10
remaining = 9
count = 1
remaining = 10
remaining = 9
count = 2
remaining = 10
End count = 2
```

控制流程(Control Flow) - Loop & While Loop

使用 while 做條件迴圈

- 在程式中用條件判斷迴圈的執行通常是很有用的。當條件為 true 時，迴圈就繼續執行。當條件不再是 true 時，程式就用 break 停止迴圈。這樣的循環行為可以用 loop、if、else 和 break 組合出來。這種模式非常常見，所以 Rust 有提供內建的結構稱為 while 迴圈。在範例中就是使用 while 讓該程式會循環五次，每次計數都減一，然後在迴圈之後印出訊息並離開：

```
/*
Output
5
4
3
2
1
done!
*/
```

```
fn main() {
    let mut counter = 5;
    while counter >= 1 {
        println!("{:?}", counter);
        counter = counter - 1;
    }
    println!("done!");
}
```

這樣消除了很多使用 loop、if、else 與 break 會有的巢狀結構，這樣可以更易閱讀。當條件為 true 的，程式碼就執行；不然的話，它就離開迴圈。

控制流程(Control Flow) - Loop & While Loop

使用 while 做條件迴圈

- 或者換個寫法：

```
fn main() {
    let mut counter = 5;
    let mut done = false;
    while !done {
        if counter > 0 {
            println!("{:?}", counter);
            counter = counter - 1;
        } else {
            done = true;
        }
    }
    println!("done!");
}
```

```
/*
Output
5
4
3
2
1
done!
*/
```

控制流程(Control Flow) - Loop & While Loop

使用 for 走訪集合

- 可以用 while 來走訪一個集合的元素，像是陣列等等。舉例來說，範例的迴圈就會印出陣列 a 的每個元素：

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("數值為：{}", a[index]);

        index += 1;
    }
}
```

```
$ cargo run
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32s
     Running `target/debug/loops`
數值為：10
數值為：20
數值為：30
數值為：40
數值為：50
```

- 程式在此對陣列的每個元素計數，它先從索引 0 開始，然後持續循環直到它抵達最後一個陣列索引為止(也就是 index < 5 不再為 true)。執行此程式會印出陣列裡的每個元素：

控制流程(Control Flow) - Loop & While Loop

使用 for 走訪集合

- 所有五個元素都如預期顯示在終端機上。儘管 index 會在某一刻達到 5，但是迴圈會在嘗試取得陣列第六個元素前就停止執行。但這樣的方式是容易出錯的，可能取得錯誤的索引數值或測試條件而造成程式恐慌。這同時也使程式變慢，因為編譯器得在執行時的程式碼對迴圈中每次疊代中進行索引是否在陣列範圍內的條件檢查。
- 所以更簡潔的替代方案是，可以使用 for 迴圈來對集合的每個元素執行一些程式碼。for 迴圈的樣子就像範例這樣：

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("數值為：{element}");
    }
}
```

控制流程(Control Flow) - Loop & While Loop

使用 for 走訪集合

- 當執行此程式時，會看到和之前範例一樣的結果。最重要的是，增加了程式的安全性，去除了造成程式錯誤的可能性。不會出現超出陣列大小或是讀取長度不足的風險。比方說在之前範例的程式碼，如果變更陣列 a 的元素為只有 4 個，但忘記更新條件判斷為 while index < 4 的話，程式就會恐慌。使用 for 迴圈的話，變更陣列長度時，就不需要去記得更新其他程式碼。

- for 迴圈的安全性與簡潔程度讓它成為 Rust 最常被使用的迴圈結構。就算想執行的是依照次數循環的程式碼，像是之前範例的 while 迴圈範例，多數Rustaceans(程式開發者)還是會選擇 for 迴圈。要這麼做的方法是使用 Range，這是標準函式庫提供的型別，用來產生一連串的數字序列，從指定一個數字開始一直到另一個數字之前結束。以下是用 for 迴圈來計數的另一種方式，它用了一個還沒講過的方法 rev，這可以用來反轉這個範圍：

```
fn main() {
    for number in (1..4).rev() {
        println!("{number}!");
    }
    println!("升空！！！");
}
```