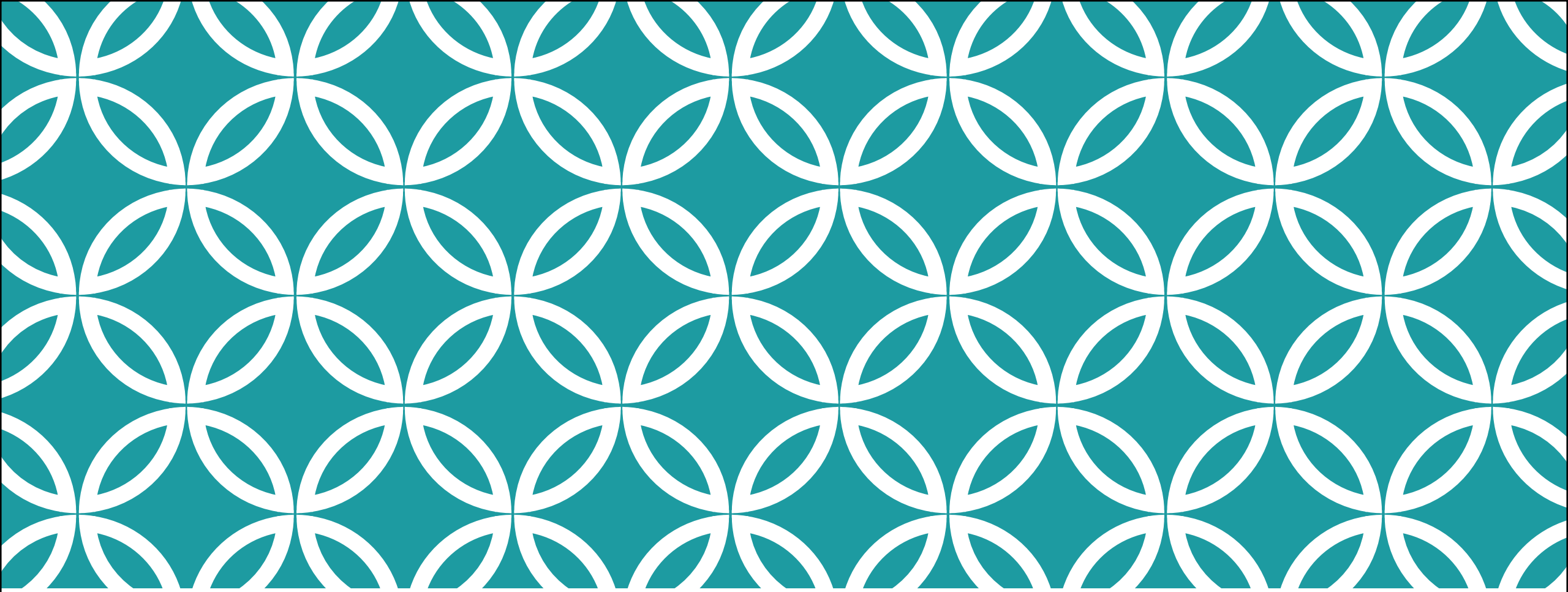

列舉與模式配對

列舉與模式配對

列舉(enumerations)有時也被簡寫為 enums。列舉定義一個能夠列舉其**可能變體(variants)的型別**。首先，會定義並使用列舉來展示列舉如何將其數據組織起來。再來會來探討一個特定的實用列舉：**Option**，其代表該值為某些東西不然就是什麼都沒有。

然後會看看 match 表達式的模式配對是怎麼運作的，讓它能夠針對列舉中不同數值執行不同的程式碼。最後會介紹 if let 這個結構，來用簡潔又方便的方式處理列舉。

列舉與模式配對

```
enum IpAddrKind {
    V4,
    V6,
}
```

定義列舉

- 結構體能將相關的欄位與資訊組織在一起，像是 Rectangle 中的 width 與 height，而列舉則是能表達一個數值屬於一組特定數值的其中一種。舉例來說，可能想要表達Rectangle是其中一種可能的形狀，而這些形狀可能還包括 Circle 與 Triangle。Rust 能以列舉的形式表現這樣的可能性。
- 看一個程式碼表達的例子，來看看為何此時用列舉會比結構體更恰當且實用。假設要使用 IP 位址，在有兩個主要的標準能使用 IP 位址：IPv4 與 IPv6。這些是程式碼可能會遇到的 IP 位址，可以列舉(enumerate)出所有可能的變體，這正是列舉的由來。
- 任何 IP 位址可以是第四版或第六版，**但不是同時存在**。IP 位址這樣的特性非常適合使用列舉資料結構，**因為列舉的值只能是其中一個變體**。第四版與第六版同時都屬於 IP 位址，所以當有程式碼要處理任何類型的 IP 位址時，它們都應該被視為相同型別。
- 要表達這樣的概念，可以定義 IpAddrKind 列舉和列出 IP 位址可能的類型 V4 和 V6 (如右上)，**這些稱為列舉的變體(variants)**。IpAddrKind 現在成了能在程式碼任何地方使用的自訂資料型別。

列舉與模式配對

```
enum IpAddrKind {
    V4,
    V6,
}
```

列舉數值

- 可以像這樣建立兩個不同變體的 IpAddrKind 實例：

```
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

- 注意變體會位於列舉命名空間底下，所以**用兩個冒號來標示**。這樣好處在於 IpAddrKind::V4和 IpAddrKind::V6 都是同型別 IpAddrKind。比方說，就可以定義一個接收任 IpAddrKind 的函式：

```
fn route(ip_kind: IpAddrKind) {}
```

- 然後可以用任意變體呼叫此函式：

```
route(IpAddrKind::V4);
route(IpAddrKind::V6);
```

列舉與模式配對

列舉數值

- 使用列舉還有更多好處。再進一步想一下 IP 位址型別還沒有辦法儲存實際的 IP 位址資料，現在只知道它是哪種類型。已經學會結構體，應該會像底下範例這樣嘗試用結構體解決問題：

```
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind,
    address: String,
}

let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};

let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
```

在這裡定義了一個有兩個欄位的結構體 IpAddr：欄位 kind 擁有 IpAddrKind(前面定義過的列舉)型別，address 欄位則是 String 型別。再來有兩個此結構體的實例。

第一個 home 擁有 IpAddrKind::V4 作為 kind 的值，然後位址資料是 127.0.0.1。第二個實例 loopback 擁有 IpAddrKind 另一個變體 V6 作為 kind 的值，且有 ::1 作為位址資料。

用結構體來組織 kind 和 address 的值在一起，讓變體可以與數值相關。

列舉與模式配對

列舉數值

- 但是可以用另一種更簡潔的方式來定義列舉就好，而不必使用結構體加上列舉。列舉內的每個變體其實都能擁有數值。以下這樣新的定義方式讓 IpAddr 的 V4 與 V6 都能擁有與其相關的 String 數值：

```
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("127.0.0.1"));

let loopback = IpAddr::V6(String::from("::1"));
```

- 將資料直接附加到列舉的每個變體上，這樣就不再用結構體。這裏還能看到另一項列舉的細節：定義的每一個列舉變體也會變成建構該列舉的函式。也就是說 IpAddr::V4() 是個函式，且接收 String 引數並回傳 IpAddr 的實例。在定義列舉時就會自動拿到這樣的建構函式。

列舉與模式配對

列舉數值

- 改使用列舉而非結構體的話還有另一項好處：每個變體可以擁有不同型別與資料的數量。V4版的 IP 位址永遠只會有四個 0 到 255 的數字部分，如果要讓 V4 儲存四個 u8，但 V6 位址仍保持 String 不變的話，在結構體是無法做到的。列舉可以輕鬆勝任：

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);

let loopback = IpAddr::V6(String::from("::1"));
```

列舉與模式配對

列舉數值

- 展示了許多種定義儲存第四版與第六版 IP 位址資料結構的方式，不過需要儲存 IP 位址並編碼成不同類型的案例實在太常見了，所以標準函式庫已經定義好了！看看標準函式庫是怎麼定義 IpAddr 的：它有一模一樣的列舉變體，不過變體各自儲存的資料是另外兩個不同的結構體，兩個定義的內容均不相同：

```
struct Ipv4Addr {
    // --省略--
}

struct Ipv6Addr {
    // --省略--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

此程式碼展示了可以將任何資料類型放入列舉的變體中：字串、數字型別、結構體都可以。甚至可以再包含另一個列舉！另外標準函式庫內的型別沒有想得那麼複雜。

列舉與模式配對

列舉數值

- 請注意雖然標準函式庫已經有定義 IpAddr，但還是可以使用並建立自己定義的型別，而且不會產生衝突，因為還沒有將標準函式庫的定義匯入到作用域中。會在之後討論如何將型別匯入作用域內。再看看底下範例的另一個列舉範例，這次的變體有各式各樣的型別：

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

此列舉有四個不同型別的變體：

1. Quit：沒有包含任何資料。
2. Move：包含了和結構體一樣的名稱欄位。
3. Write：包含了一個 String。
4. ChangeColor：包含了三個 i32。

列舉與模式配對

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

列舉數值

- 如同上面範例這樣定義列舉變體和定義不同類型的結構體很像，只不過列舉不使用 struct 關鍵字，而且所有的變體都會在 Message 型別底下。以下的結構體可以包含與上方列舉變體定義過的資料：

```
struct QuitMessage; // 類單元結構體
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // 元組結構體
struct ChangeColorMessage(i32, i32, i32); // 元組結構體
```

- 但是**如果使用不同結構體且各自都有自己的型別的話，就無法像之前範例那樣將 Message 視為單一型別，輕鬆在定義函式時接收訊息所有可能的類型。**

列舉與模式配對

列舉數值

- 列舉和結構體還有一個地方很像：如同可以對結構體使用 impl 定義方法，也可以對列舉定義方法。以下範例顯示可以對 Message 列舉定義一個 call 方法：

```
impl Message {
    fn call(&self) {
        // 在此定義方法本體
    }
}

let m = Message::Write(String::from("hello"));
m.call();
```

- 方法本體使用 self 來取得呼叫方法的值。在此例中，建立了一個變數 m 並取得 Message::Write(String::from("hello"))，而這就會是當執行 m.call() 時 call 方法內會用到的 self。
- 再看看另一個標準函式庫內非常常見且實用的列舉：**Option**。

列舉與模式配對

Option 列舉相對於空值的優勢

- 在此來研究 Option，這是在標準函式庫中定義的另一種列舉。Option 廣泛運用在許多場合，**它能表示一個數值可能有某個東西或者什麼都沒有。**
- 舉例來說，如果向一串包含元素的列表索取第一個值，會拿到數值，但如果向空列表索取的話，就什麼都拿不到。在型別系統中表達這樣的概念可以讓編譯器檢查是否都處理完該處理的情況了。這樣的功能可以防止其他程式語言中極度常見的程式錯誤。
- 程式語言設計通常要考慮哪些功能是要的，但同時哪些功能是不要的也很重要。**Rust 沒有像其他許多語言都有空值**。空值(Null)代表的是沒有任何數值。在有空值的語言，所有變數都有兩種可能：空值或非空值。

列舉與模式配對

Option 列舉相對於空值的優勢

- 空值的問題在於，如果想在非空值使用空值的話，會得到某種錯誤。由於空值與非空值的特性無所不在，會很容易犯下這類型的錯誤。但有時候能夠表達「空(null)」的概念還是很有用的：**空值代表目前的數值因為某些原因而無效或缺少。**
- 所以問題不在於概念本身，而在於如何實作。所以 Rust 並沒有空值，但是它有一個列舉可以表達出這樣的概念，也就是一個值可能是存在或不存在的。此列舉就是 Option<T>，它是在標準函式庫中這樣定義的：

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

Option<T> 實在太實用了，所以它早已加進 prelude 中，不需要特地匯入作用域中。它的變體一樣也被加進 prelude 中，可以直接使用 Some 和 None 而不必加上 Option:: 的前綴。Option<T> 仍然就只是個列舉， Some(T) 與 None 仍然是Option<T> 型別的變體。

列舉與模式配對

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

Option 列舉相對於空值的優勢

- <T> 語法是還沒介紹到的 Rust 功能。它是個泛型型別參數，會在之後正式介紹泛型(generics)。現在只需要知道 <T> 指的是 Option 列舉中的 Some 變體可以是任意型別。而透過 Option 數值來持有數字型別和字串型別的話，它們最終會換掉 Option<T> 中的 T，成為不同的型別。以下是使用 Option 來包含數字與字串型別的範例：

```
let some_number = Some(5);
let some_char = Some('e');

let absent_number: Option<i32> = None;
```

- some_number 的型別是 Option<i32>，而 some_char 的型別是 Option<char>，兩者是不同的型別。Rust 可以推導出這些型別，因為已經在 Some 變體指定數值。至於 absent_number 的話，Rust 需要寫出完整的Option型別，因為編譯器無法從None推導出相對應的 Some 變體會持有哪種型別。要在這裡告訴 Rust， absent_number 所指的型別為 Option<i32>。

列舉與模式配對

```rust
let some_number = Some(5);
let some_char = Some('e');

let absent_number: Option<i32> = None;
```

Option 列舉相對於空值的優勢

- 當有 Some 值時，會知道數值是存在的而且就位於 Some 內。當有 None 值時，在某種意義上它代表該值是空的，沒有有效的數值。所以為何 Option<T> 會比用空值來得好呢？
- 簡單來說因為Option<T>與T(T可以是任意型別)是不同的型別，編譯器不會允許像一般有效的值那樣來使用 Option<T>。舉例來說，**以下範例是無法編譯的**，因為這是將 i8 與 Option<i8> 相加：

```rust
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0277]: cannot add `Option<i8>` to `i8`
 --> src/main.rs:5:17
  |
5 |     let sum = x + y;
  |                 ^ no implementation for `i8 + Option<i8>`
  |
  = help: the trait `std::ops::Add<Option<i8>>` is not implemented for `i8`
  = help: the following other types implement trait `Add<Rhs>`:
            <&'a f32 as Add<f32>>
            <&'a f64 as Add<f64>>
            <&'a i128 as Add<i128>>
            <&'a i16 as Add<i16>>
            <&'a i32 as Add<i32>>
            <&'a i64 as Add<i64>>
            <&'a i8 as Add<i8>>
            <&'a isize as Add<isize>>
          and 48 others

For more information about this error, try `rustc --explain E0277`.
error: could not compile `enums` due to previous error
```

列舉與模式配對

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0277]: cannot add `Option<i8>` to `i8`
 --> src/main.rs:5:17
  |
5 |     let sum = x + y;
  |                 ^ no implementation for `i8 + Option<i8>`
  |
  = help: the trait `std::ops::Add<Option<i8>>` is not implemented for `i8`
  = help: the following other types implement trait `Add<Rhs>`:
            <&'a f32 as Add<f32>>
            <&'a f64 as Add<f64>>
            <&'a i128 as Add<i128>>
            <&'a i16 as Add<i16>>
            <&'a i32 as Add<i32>>
            <&'a i64 as Add<i64>>
            <&'a i8 as Add<i8>>
            <&'a isize as Add<isize>>
          and 48 others

For more information about this error, try `rustc --explain E0277`.
error: could not compile `enums` due to previous error
```

Option 列舉相對於空值的優勢

- 這樣其實很好！此錯誤訊息事實上指的是 Rust 不知道如何將 i8 與 Option<i8> 相加，因為它們是不同的型別。當在 Rust 中有個型別像是 i8，編譯器將會確保永遠會擁有有效數值。
- 可以很放心地使用該值，而不必檢查是不是空的。只有在使用 Option<i8>(或者任何其他要使用的型別)時才需要去擔心會不會沒有值。然後編譯器會確保在使用該值前，有處理過該有的條件。換句話說，必須將Option<T>轉換為T才能對T做運算。這通常就能幫助抓到空值最常見的問題：認為某值不為空，但它其實就是空值。
- 消除掉非空值是否正確的風險，可以對寫的程式碼更有信心。**要讓一個值變成可能為空的話，必須顯式建立成對應型別的 Option<T>**。然後當要使用該值時，就得顯式處理數值是否為空的條件。只要一個數值的型別不是 Option<T>，就可以安全地認定該值不為空。**這是 Rust 刻意考慮的設計決定，限制無所不在的空值，並增強 Rust 程式碼的安全性。**

列舉與模式配對

Option 列舉相對於空值的優勢

- 所以當有一個數值型別 Option<T>，要怎麼從 Some 變體取出 T，好可以使用該值呢？Option<T> 列舉有大量實用的方法可以在不同的場合下使用。可以在它的技術文件查閱。更加熟悉 Option<T> 的方法十分益於接下來的 Rust 旅程。
- 整體來說，要使用 Option<T> 數值的話，要讓程式碼可以處理每個變體：
  1. 會希望有一些程式碼只會在當有 Some(T) 時執行，然後這些程式碼允許使用內部的 T。
  2. 會希望有另一部分的程式碼能在只有 None 時執行，且這些程式碼不會拿到有效的 T 數值。
- **match 表達式正是處理此列舉行為的控制流結構：它會針對不同的列舉變體執行不同的程式碼，而且程式碼可以使用配對到的數值資料。**

列舉與模式配對

match 控制流建構子

- Rust 有個功能非常強大的控制流建構子叫做 match，可以使用一系列模式來配對數值並依據配對到的模式來執行對應的程式。模式(Patterns)可以是字面數值、變數名稱、萬用字元(wildcards)和其他更多元件來組成。
- 可以想像 match 表達式成一個硬幣分類機器：硬幣會滑到不同大小的軌道，然後每個硬幣會滑入第一個符合大小的軌道。同樣地，數值會依序走訪 match 的每個模式，然後進入第一個「配對」到該數值的模式所在的程式碼區塊，並在執行過程中使用。

列舉與模式配對

match 控制流建構子

- 既然都提到硬幣了，就用它們來作為 match 的範例吧！可以寫一個接收未知美國硬幣的函式，以類似驗鈔機的方式，決定它是何種硬幣並以美分作為單位回傳其值。如底下範例所示：

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

value_in_cents 函式中 match 的每個部分。首先使用 match 並加上一個表達式，在此例的話就是指 coin。這和 if 中條件表達式的用法很像。**不過差別在於 if 中的條件必須是布林值，而在此它可以回傳任何型別**。在此範例中 coin 的型別是在第一行定義的列舉 Coin。接下來是 match 的分支，每個分支有兩個部分：一個模式以及對應的程式碼。這邊第一個分支的模式是 Coin::Penny 然後 => 會將模式與要執行的程式碼分開來，而在此例的程式碼就只是個 1。每個分支之間由逗號區隔開來。

列舉與模式配對

match 控制流建構子

- 既然都提到硬幣了，就用它們來作為 match 的範例吧！可以寫一個接收未知美國硬幣的函式，以類似驗鈔機的方式，決定它是何種硬幣並以美分作為單位回傳其值。如底下範例所示：

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

當 match 表達式執行時，會將計算的數據結果依序與每個分支的模式做比較。如果有模式配對到該值的話，其對應的程式碼就會執行。如果該模式與數值不符的話，就繼續執行下一個分支，就像硬幣分類機器。

每個分支對應的程式碼都是表達式，然後在配對到的分支中表達式的數值結果就會是整個 match 表達式的回傳值。

列舉與模式配對

match 控制流建構子

- 如果配對分支的程式碼很短的話，通常就不會用到大括號，像是底下範例的每個分支就只回傳一個數值。如果想要在配對分支執行多行程式碼的話，就必須用大括號，然後可以在括號後選擇性加上逗號。舉例來說，以下程式會在每次配對到 Coin::Penny 時印出「幸運幣！」再回傳程式碼區塊最後的數值 1：

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("幸運幣！");
            1
        }
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

列舉與模式配對

綁定數值的模式

- 另一項配對分支的實用功能是可以綁定配對模式中部分的數值，這可以取出列舉變體中的數值。
- 舉例來說，改變其中一個列舉變體成擁有資料。從1999年到2008年，美國在鑄造25美分硬幣時，其中一側會有 50 個州不同的設計。不過其他的硬幣就沒有這樣的設計，只有 25 美分會有特殊值而已。可以改變 enum 中的 Quarter 變體成儲存 UsState 數值，如底下範例所示：

```
#[derive(Debug)] // 這讓我們可以顯示每個州
enum UsState {
    Alabama,
    Alaska,
    // --省略--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

想像有一個朋友想要收集所有 50 州的 25 美分硬幣。當在排序零錢的同時，會在拿到 25 美分時喊出該硬幣對應的州好讓朋友知道，如果沒有的話就可以納入收藏。

列舉與模式配對

```rust
#[derive(Debug)] // 這讓我們可以顯示每個州
enum UsState {
    Alabama,
    Alaska,
    // --省略--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

綁定數值的模式

- 在此程式中的配對表達式中，在 Coin::Quarter 變體的配對模式中新增了一個變數 state。當 Coin::Quarter 配對符合時，變數 state 會綁定該 25 美分的數值，然後就可以在分支程式碼中使用 state，如以下所示：

```rust
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("此 25 美分所屬的州為 {:?}!", state);
            25
        }
    }
}
```

如果呼叫 value_in_cents(Coin::Quarter(UsState::Alaska)) 的話，coin 就會是 Coin::Quarter(UsState::Alaska)。

當比較每個配對分支時，會到 Coin::Quarter(state) 的分支才配對成功。此時 state 綁定的數值就會是 UsState::Alaska。就可以在 println! 表達式中使用該綁定的值，以此取得 Coin 列舉中 Quarter 變體內的值。

列舉與模式配對

配對 Option<T>

- 在前一個段落，想要在使用 Option<T> 時取得 Some 內部的 T 值。如同列舉 Coin，一樣可以使用 match 來處理 Option<T>！相對於比較硬幣，要比較的是 Option<T> 的變體，不過 match 表達式運作的方式一模一樣。
- 假設要寫個接受 Option<i32> 的函式，而且如果內部有值的話就將其加上 1。如果內部沒有數值的話，該函式就回傳 None 且不再嘗試做任何動作。拜 match 所賜，這樣的函式很容易寫出來，長得就像底下範例：

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

列舉與模式配對

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

配對 Option<T>

- 來仔細分析 plus_one 第一次的執行結果。當呼叫 plus_one(five) 時，plus_one 本體中的變數 x 會擁有 Some(5)。接著就拿去和每個配對分支比較：`None => None,`
- Some(5) 並不符合 None 這樣的模式，所以繼續進行下一個分支：`Some(i) => Some(i + 1),`
- Some(5) 有符合 Some(i) 這樣的模式嗎？這是當然的囉！有相同的變體。i 會綁定 Some 中的值，所以 i 會取得 5。接下來配對分支中的程式碼就會執行，將 1 加入 i 並產生新的 Some 其內部的值就會是 6。
- 現在讓看看範例中第二次的 plus_one 呼叫，這次的 x 是 None。進入match然後比較第一個分支：

```
None => None,
```

- 配對成功！因為沒有任何數值可以相加，程式就停止並在 => 之後馬上回傳 None。因為第一個分支就配對成功了，沒有其他的分支需要再做比較。

列舉與模式配對

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```

配對必須是徹底的

- 還有一個 match 的細節要討論：分支的模式必須涵蓋所有可能性。要是像這樣寫了一個有錯誤的 plus_one 函式版本，它會無法編譯：
- 沒有處理到 None 的情形，所以此程式碼會產生錯誤。幸運的是這是 Rust 能夠抓到的錯誤。如果嘗試編譯此程式的話，會得到以下錯誤：

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0004]: non-exhaustive patterns: `None` not covered
 --> src/main.rs:3:15
  |
3 |         match x {
  |               ^ pattern `None` not covered
  |
note: `Option<i32>` defined here
  = note: the matched value is of type `Option<i32>`
help: ensure that all possible cases are being handled by adding a match arm with a wild
  |
4 ~             Some(i) => Some(i + 1),
5 ~             None => todo!(),
  |

For more information about this error, try `rustc --explain E0004`.
error: could not compile `enums` due to previous error
```

Rust 發現沒有考慮到所有可能條件，而且還知道少了哪些模式！Rust 中的配對必須是徹底(exhaustive)的：必須列舉出所有可能的情形，程式碼才能夠被視為有效。

尤其是在 Option<T> 的情況下，當 Rust 防止忘記處理 None 的情形時，它也免於以為擁有一個有效實際上卻是空的值。

列舉與模式配對

Catch-all 模式與 _ 佔位符

- 使用列舉的話，可以針對特定數值作特別的動作，而對其他所有數值採取預設動作。
- 想像一下正在做款骰子遊戲，如果骰出 3 的話，角色就動不了，但是可以拿頂酷炫的帽子。如果骰出 7，角色就損失那頂帽子。至於其他的數值，角色就按照那個數值在遊戲桌上移動步數。以下是用 match 實作出的邏輯，骰子的結果並非隨機數而是寫死的，且所有邏輯對應的函式本體都是空的，因為實際去實作並非本範例的重點：

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    other => move_player(other),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn move_player(num_spaces: u8) {}
```

列舉與模式配對

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    other => move_player(other),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn move_player(num_spaces: u8) {}
```

Catch-all 模式與 _ 佔位符

- 在前兩個分支中的模式分別為數值 3 和 7。至於最後一個涵蓋其他可能數值的分支，用變數 other 作為模式。在 other 分支執行的程式碼會將該變數傳入函式 move_player 中。
- 此程式碼就算沒有列完所有 u8 可能的數字也能編譯完成，因為最後的模式會配對所有尚未被列出來的數值。**這樣的catch-all模式能滿足match必須要徹底的要求**。注意到需要將catch-all分支放在最後面，因為模式是按照順序配對的。如果將 catch-all 放在其他分支前的話，這樣一來其他後面的分支就永遠配對不到了，所以要是在 catch-all 之後仍加上分支的話 Rust 會警告！
- 當想使用 catch-all 模式但不想使用其數值時，Rust 還有一種模式能使用：**_ 這是個特殊模式**，用來配對任意數值且不綁定該數值。這告訴Rust不會用到該數值，所以不會警告沒使用到變數。

列舉與模式配對

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => reroll(),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn reroll() {}
```

Catch-all 模式與 _ 佔位符

- 來改變一下遊戲規則：如果骰到除了 3 與 7 以外的話，必須要重新擲骰。不需要用到 catch-all 的數值，所以可以修改程式碼來使用 _，而不必繼續用變數 other：
- 此範例一樣也滿足徹底的要求，因為在最後的分支顯式地忽略其他所有數值，沒有遺漏任何值。
- 再改一次遊戲規則，改成如果骰到除了 3 與 7 以外，不會有任何事發生的話，**可以用單元數值(在元組型別段落提到的空元組)作為 _ 的程式碼**：

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => (),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
```

這裡顯式地告訴 Rust 不會使用任何其他沒被先前分支配對到的數值，而且也不想在此執行任何程式碼。

列舉與模式配對

Match 表達式(Expression)

- match 的許多優點之一是「徹底檢查」(exhaustiveness checking)
- **舉例來說，當刪除最後一條有著 _ 的執行分支，編譯器將會給錯誤訊息**

```
let a = 2;

match a {
    1 => println!("一"),
    2 => println!("二"),
    3 => println!("三"),
    4 => println!("四"),
    5 => println!("五"),
    _ => println!("其他"),
}
```

列舉與模式配對

Boolean Match

```
fn main() {
    let is_valid = true;
    match is_valid {
      true => println!("Yeaaa!"),
      false => println!("OOPS!"),
    }
}
```

Number Match

```
fn main() {
    let number = 100;
    match number {
        0 => println!("Zero!"),
        1 | 3 | 5 | 7 | 11 => println!("This is odd!"),
        2 | 4 | 6 | 8 | 10 => println!("This is even!"),
        12..=18 => println!("Less than eighteen."),
        _ => println!("Ain't special"),
    }
}
```

列舉與模式配對

透過 if let 簡化控制流

- if let 語法可以用 if 與 let 的組合來以比較不冗長的方式，來處理只在乎其中一種模式而忽略其餘的數值。現在考慮一支程式如底下範例所示，在配對 config_max 中 Option<u8> 的值，但只想在數值為 Some 變體時執行程式：

```
let config_max = Some(3u8);
match config_max {
    Some(max) => println!("最大值被設為 {}", max),
    _ => (),
}
```

- 如果數值為 Some，就在分支中綁定 max 變數，印出 Some 變體內的數值。不想對 None 作任何事情。為了滿足 match 表達式，必須在只處理一種變體的分支後面，再加上 _ => ()。這樣就加了不少樣板程式碼。

列舉與模式配對

```
let config_max = Some(3u8);
match config_max {
    Some(max) => println!("最大值被設為 {}", max),
    _ => (),
}
```

透過 if let 簡化控制流

- 不過可以使用 if let 以更精簡的方式寫出來，以下程式碼的行為就與之前範例的 match 一樣：

```
let config_max = Some(3u8);
if let Some(max) = config_max {
    println!("最大值被設為 {}", max);
}
```

- if let 接收一個模式與一個表達式，然後用等號區隔開來。它與 match 的運作方式相同，表達式的意義與match相同，然後前面的模式就是第一個分支。 在此例中的模式就是 Some(max)，然後 max 會綁定 Some 內的數值。
- 就和 match 分支中使用 max 一樣，在 if let 區塊的本體中使用 max。如果數值沒有配對到模式，if let 中的程式碼就不會執行。**使用 if let 可以少打些字、減少縮排以及不用寫多餘的樣板程式碼**。不過就少了 match 強制的徹底窮舉檢查。要何時選擇 match 還是 if let 得依據在的場合是要做什麼事情，以及在精簡度與徹底檢查之間做取捨。
- 換句話說，可以想像 if let 是 match 的語法糖(syntax sugar)，它只會配對一種模式來執行程式碼並忽略其他數值。

列舉與模式配對

透過 if let 簡化控制流

- 也可以在 if let 之後加上else，else 之後的程式碼區塊等同於match表達式中 _ 情形的程式碼區塊。這樣一來的 if let 和 else 組合就等同於 match 了。回想一下之前範例的 Coin 列舉定義，Quarter 變體擁有數值 UsState。如果希望統計所有不是 25 美分的硬幣的同時，也能繼續回報 25 美分所屬的州，可以用 match 像這樣寫：

```
let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("此 25 美分所屬的州為 {:?}!", state),
    _ => count += 1,
}
```

- 或是也可以用 if let 和 else 表達式這樣寫：

```
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("此 25 美分所屬的州為 {:?}!", state);
} else {
    count += 1;
}
```

- 如果程式碼邏輯遇到使用 match 表達會太囉唆的話，記得 if let 也在的 Rust 工具箱中供使用。