錯誤處理

錯誤處理

錯誤是軟體開發中不可避免的一環，所以Rust有一些特色能夠處理發生錯誤的情形。在許多情況下，Rust 要求要能知道可能出錯的地方，並在編譯前採取行動。這樣的要求能讓程式更穩定，確保能發現錯誤並在程式碼發佈到生產環境前妥善處理它們！

Rust 將錯誤分成兩大類：**可復原的(recoverable)和不可復原的(unrecoverable)錯誤**。像是找不到檔案這種可復原的錯誤，通常很可能只想回報問題給使用者並重試。而不可復原的錯誤就會是程式錯誤的跡象，像是嘗試取得陣列結尾之後的位置。

許多語言不會區分這兩種錯誤，並以相同的方式處理，使用像是例外(exceptions)這樣統一的機制處理。**Rust 沒有例外處理機制，取而代之的是它對可復原的錯誤提供 Result<T, E> 型別，對不可復原的錯誤使用 panic! 將程式停止執行**。會先介紹 panic! 再來討論 Result<T, E> 數值的回傳。除此之外，也將探討何時該從錯誤中復原，何時該選擇停止程式。

錯誤處理

對無法復原的錯誤使用 panic!

- 有時候壞事就是會發生在程式中，這本來就是沒辦法全部避免的。在這種情況，Rust有提供panic! 巨集。在實際情況下有兩種方式可以造成恐慌：**做出確定會讓程式碼恐慌的動作(像是存取陣列範圍外的元素)或是直接呼叫 panic! 巨集**。
- 在這兩種狀況下，都對程式造成了恐慌。這些恐慌預設會印出程式出錯的訊息，展開並清理堆疊，然後離開程式。再加上環境變數的話，還可以讓 Rust 顯示恐慌時呼叫的堆疊，能更簡單地追蹤恐慌的源頭。

錯誤處理

恐慌時該解開堆疊還是直接終止

- 當恐慌(panic)發生時，程式預設會開始做**解開(unwind)**堆疊的動作，這代表 Rust 會回溯整個堆疊，並清理每個它遇到的函式資料。但是這樣回溯並清理的動作很花力氣。另一種方式是直接**終止(abort)**程式而不清理，程式使用的記憶體會需要由作業系統來清理。
- 如果需要專案產生的執行檔越小越好，可以從解開切換成終止，只要在 Cargo.toml 檔案中的 [profile] 段落加上 panic = 'abort' 就好。舉例來說，如果希望在發佈模式(release mode)恐慌時直接終止，那就加上：

```
[profile.release]
panic = 'abort'
```

錯誤處理

對無法復原的錯誤使用 panic!

- 先在小程式內試試呼叫 panic!：

```
fn main() {
    panic!("◢▆▅▃ 崩╰(〒皿〒)╯潰▃▅▆◣");
}
```

- 當執行程式時，會看到像這樣的結果：

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s
     Running `target/debug/panic`
thread 'main' panicked at '◢▆▅▃ 崩╰(〒皿〒)╯潰▃▅▆◣', src/main.rs:2:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

- panic!的呼叫導致印出最後兩行的錯誤訊息。第一行顯示恐慌訊息以及該恐慌是在原始碼何處發生的：src/main.rs:2:5 指的是它發生在 src/main.rs 檔案第二行第五個字元。
- 在此例中，該行指的就是寫的程式碼。如果查看該行，會看到 panic! 巨集的呼叫。在其他情形，panic! 的呼叫可能會發生在呼叫的其他程式碼內，所以錯誤訊息回報的檔案名稱與行數可能就會是其他人呼叫 panic! 巨集的程式碼，而不是因為程式碼才導致 panic! 的呼叫。可以在呼叫 panic! 程式碼的地方使用 backtrace 來找出出現問題的地方。接下來就會深入瞭解 backtrace。

錯誤處理

對無法復原的錯誤使用 panic!

- 使用 panic! Backtrace
  - 看看另一個例子，這是函式庫發生錯誤而呼叫 panic!，而不是來自於程式碼自己呼叫的巨集。底下範例是個嘗試從向量有效範圍外取得索引的例子：

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

    v[99];
}
```

  - 在這邊嘗試取得向量中第 100 個元素(不過因為索引從零開始，所以是索引 99)，但是該向量只有 3 個元素。在此情況下，Rust 就會恐慌。使用 [] 會回傳元素，但是如果傳遞了無效的索引，Rust 就回傳不了正確的元素。
  - 在 C 中，嘗試讀取資料結構結束之後的元素屬於未定義行為。可能會得到該記憶體位置對應其資料結構的元素，即使該記憶體完全不屬於該資料結構。這就稱做**緩衝區過讀(buffer overread)**而且會導致安全漏洞。
  - 攻擊者可能故意操縱該索引來取得在資料結構後面原本不應該讀寫的值。

錯誤處理

對無法復原的錯誤使用 panic!

- 使用 panic! Backtrace
  - 為了保護程式免於這樣的漏洞，如果嘗試用一個不存在的索引讀取元素的話，Rust 會停止執行並拒絕繼續運作下去。嘗試執行並看看會如何：

```
$ cargo run
   Compiling panic v0.1.0 (file:///projects/panic)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/panic`
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99', src/
main.rs:4:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

  - 此錯誤指向main.rs的第四行，也就是嘗試存取索引99的地方。
  - 下一行提示告訴可以設置 RUST_BACKTRACE環境變數來取得 backtrace 以知道錯誤發生時到底發生什麼事

錯誤處理

對無法復原的錯誤使用 panic!

```
$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99', src/main.rs:4:5
stack backtrace:
   0: rust_begin_unwind
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/std/src/panicking.rs:584:5
   1: core::panicking::panic_fmt
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/panicking.rs:142:14
   2: core::panicking::panic_bounds_check
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/panicking.rs:84:5
   3: <usize as core::slice::index::SliceIndex<[T]>>::index
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/slice/index.rs:242:10
   4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/slice/index.rs:18:9
   5: <alloc::vec::Vec<T,A> as core::ops::index::Index<I>>::index
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/alloc/src/vec/mod.rs:2591:9
   6: panic::main
             at ./src/main.rs:4:5
   7: core::ops::function::FnOnce::call_once
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/ops/function.rs:248:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

- 使用 panic! Backtrace
  - backtrace 是一個函式列表，指出得到此錯誤時到底依序呼叫了哪些函式。
  - Rust 的 backtraces 運作方式和其他語言一樣：讀取 backtrace 關鍵是從最一開始讀取直到看到寫的檔案。那就會是問題發生的源頭。那行以上的行數就是所呼叫的程式，而以下則是其他呼叫程式碼的程式。這些行數可能還會包含Rust核心程式碼、標準函式庫程式碼，或是所使用的crate。
  - 設置RUST_BACKTRACE環境變數的值不為 0，來嘗試取得backtrace吧。會看到和底下範例類似的結果：

錯誤處理

對無法復原的錯誤使用 panic!

```
$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99', src/main.rs:4:5
stack backtrace:
   0: rust_begin_unwind
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/std/src/panicking.rs:584:5
   1: core::panicking::panic_fmt
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/panicking.rs:142:14
   2: core::panicking::panic_bounds_check
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/panicking.rs:84:5
   3: <usize as core::slice::index::SliceIndex<[T]>>::index
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/slice/index.rs:242:10
   4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/slice/index.rs:18:9
   5: <alloc::vec::Vec<T,A> as core::ops::index::Index<I>>::index
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/alloc/src/vec/mod.rs:2591:9
   6: panic::main
             at ./src/main.rs:4:5
   7: core::ops::function::FnOnce::call_once
             at /rustc/e092d0b6b43f2de967af0887873151bb1c0b18d3/library/core/src/ops/function.rs:248:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

- 使用 panic! Backtrace
  - 輸出結果有點多啊！看到的實際輸出可能會因作業系統與 Rust 版本而有所不同。
  - 要取得這些資訊的 backtrace，除錯符號(debug symbols)必須啟用。當在使用 cargo build 或 cargo run 且沒有加上 --release 時，除錯符號預設是啟用的。
  - 在上面範例的輸出結果中，第 6 行的 backtrace 指向了專案中產生問題的地方：src/main.rs 中的第四行。如果不想讓程式恐慌，就要來調查所寫的程式中第一個被錯誤訊息指向的位置。
  - 在前面範例中，故意寫出會恐慌的程式碼。要修正的方法就是不要索取超出向量索引範圍的元素。當在未來的程式碼恐慌時，會需要知道是程式碼中的什麼動作造成的、什麼數值導致恐慌以及正確的程式碼該怎麼處理。
  - **在「要 panic! 還是不要 panic!」的段落中再回來看 panic! 並研究何時該與不該使用 panic! 來處理錯誤條件。**接下來，要看如何使用 Result來處理可復原的錯誤。

錯誤處理

Result 與可復原的錯誤

- 大多數的錯誤沒有嚴重到需要讓整個程式停止執行。有時候當函式失敗時，是可以輕易理解並作出反應的。舉例來說，如果嘗試開啟一個檔案，但該動作卻因為沒有該檔案而失敗的話，可能會想要建立檔案，而不是終止程序。
- 回憶一下之前的「使用 Result 型別可能的錯誤」提到 Result 列舉的定義有兩個變體 Ok 和 Err，如以下所示：

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

- T 和 E 是泛型型別參數，會在之後深入討論泛型。**現在需要知道的是 T 代表在成功時會在 Ok 變體回傳的型別，而 E 則代表失敗時在 Err 變體會回傳的錯誤型別**。因為 Result 有這些泛型型別參數，可以將 Result 型別和它的函式用在許多不同場合，讓成功與失敗時回傳的型別不相同。

錯誤處理

Result 與可復原的錯誤

- 呼叫一個可能會失敗的函式並回傳 Result 型別。在底下範例嘗試開啟一個檔案：

```
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");
}
```

- File::open 的回傳型別為 Result<T, E>。泛型參數 T 在此已經被 File::open 指明成功時會用到的型別 std::fs::File，也就是檔案的控制代碼(handle)。用於錯誤時的 E 型別則是 std::io::Error。這樣的回傳型別代表 File::open 的呼叫在成功時會回傳可以讀寫的檔案控制代碼，但該函式呼叫也可能失敗。
- 舉例來說，該檔案可能會不存在，或者沒有檔案的存取權限。File::open 需要有某種方式能告訴它的結果是成功或失敗，並回傳檔案控制代碼或是錯誤資訊。這樣的資訊正是 Result 列舉想表達的。

錯誤處理

Result 與可復原的錯誤

```
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");
}
```

- 如果 File::open 成功的話，變數 greeting_file_result 的數值就會獲得包含檔案控制代碼的 Ok 實例。如果失敗的話，greeting_file_result 的值就會是包含為何產生該錯誤的資訊的 Err 實例。
- 需要讓範例的程式碼依據 File::open 回傳不同的結果採取不同的動作。底下範例展示了其中一種處理 Result 的方式，使用之前提到的 match 表達式：

```
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => panic!("開啟檔案時發生問題：{:?}", error),
    };
}
```

和 Option 列舉一樣，Result 列舉與其變體都會透過 prelude 引入作用域，所以不需要指明 Result::，可以直接在 match 的分支中使用 Ok 和 Err 變體。

當結果是 Ok 時，這裡的程式碼就會回傳 Ok 變體中內部的 file，然後就可以將檔案控制代碼賦值給變數 greeting_file。在 match 之後，就可以使用檔案控制程式碼來讀寫。

錯誤處理

Result 與可復原的錯誤

```rust
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => panic!("開啟檔案時發生問題：{:?}", error),
    };
}
```

- match 的另一個分支則負責處理從File::open中取得的Err數值。在此範例中，選擇呼叫 panic! 巨集。如果檔案 hello.txt 不存在當前的目錄的話，就會執行此程式碼，接著就會看到來自 panic! 巨集的輸出結果：

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/error-handling`
thread 'main' panicked at '開啟檔案時發生問題：Os { code: 2, kind: NotFound, message: "No such file or
directory" }', src/main.rs:8:23
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

- 如往常一樣，此輸出告訴哪裡出錯了。

錯誤處理

Result 與可復原的錯誤

- 配對不同種類的錯誤

```rust
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => panic!("開啟檔案時發生問題：{:?}", error),
    };
}
```

  - 上面範例的程式碼不管 File::open 為何失敗都會呼叫 panic!。不過想要依據不同的錯誤原因採取不同的動作，如果 File::open 是因為檔案不存在的話，想要建立檔案並回傳新檔案的控制代碼；如果 File::open 是因為其他原因失敗的話，像是沒有開啟檔案的權限，仍然要像範例這樣呼叫 panic!。對此可以加上 match 表達式，如底下範例所示：

```rust
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("建立檔案時發生問題：{:?}", e),
            },
            other_error => {
                panic!("開啟檔案時發生問題：{:?}", other_error);
            }
        },
    };
}
```

錯誤處理

Result 與可復原的錯誤

- 配對不同種類的錯誤

```rust
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("建立檔案時發生問題：{:?}", e),
            },
            other_error => {
                panic!("開啟檔案時發生問題：{:?}", other_error);
            }
        },
    };
}
```

  - File::open 在 Err 變體的回傳型別為 io::Error，這是標準函式庫提供的結構體。此結構體有個 kind 方法可以取得 io::ErrorKind 數值。標準函式庫提供的列舉 io::ErrorKind 有從 io 運算可能發生的各種錯誤。想處理的變體是 ErrorKind::NotFound，這指的是嘗試開啟的檔案還不存在。所以對 greeting_file_result 配對並在用 error.kind() 繼續配對下去。
  - 從內部配對檢查 error.kind() 的回傳值是否是 ErrorKind 列舉中的 NotFound 變體。如果是的話，就嘗試使用 File::create 建立檔案。不過 File::create 也可能會失敗，所以需要第二個內部 match 表達式來處理。如果檔案無法建立的話，就會印出不同的錯誤訊息。第二個分支的外部 match 分支保持不變，如果程式遇到其他錯誤的話就會恐慌。

錯誤處理

除了使用 match 配對 Result<T, E> 以外的方式

- 用的 match 的確有點多！match 表達式雖然很實用，不過它的行為非常基本。在之後會學到閉包(closure)，Result<T, E> 型別有很多接收閉包的方法。**使用這些方法可以讓程式碼更簡潔**。
- 舉例來說，以下是另一種能寫出與之前範例邏輯相同的程式碼，這次則是使用到閉包與 unwrap_or_else 方法：

```
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("建立檔案時發生問題：{:?}", error);
            })
        } else {
            panic!("開啟檔案時發生問題：{:?}", error);
        }
    });
}
```

- 雖然此程式碼的行為和之前範例一樣，但沒有包含任何 match 表達式而且更易閱讀。

錯誤處理

Result 與可復原的錯誤

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap();
}
```

- 錯誤發生時產生恐慌的捷徑：unwrap 與 expect
  - 雖然 match 已經足以勝任指派的任務了，但它還是有點冗長，而且可能無法正確傳遞錯誤的嚴重性。Result<T, E> 型別有非常多的輔助方法來執行不同的特定任務。
  - unwrap 就和之前範例所寫的 match 表達式一樣，擁有類似效果的捷徑方法。如果 Result 的值是 Ok 變體，unwrap會回傳 Ok 裡面的值；如果 Result 是 Err 變體的話，unwrap 會呼叫 panic! 巨集。右上是使用 unwrap 的方式：
  - 如果沒有 hello.txt 這個檔案並執行此程式碼的話，會看到從 unwrap 方法所呼叫的 panic! 回傳訊息：

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: Os {
code: 2, kind: NotFound, message: "No such file or directory" }',
src/main.rs:4:49
```

錯誤處理

Result 與可復原的錯誤

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")
        .expect("hello.txt 應該要存在此專案中");
}
```

- 錯誤發生時產生恐慌的捷徑：unwrap 與 expect
  - 還有另一個方法 expect 和 unwrap 類似，不過**能選擇 panic! 回傳的錯誤訊息**。使用 expect 而非 unwrap 並提供完善的錯誤訊息可以表明意圖，讓追蹤恐慌的源頭更容易。expect 的語法看起來就像右上這樣：
  - 使用 expect 的方式和 unwrap 一樣，不是回傳檔案控制代碼就是呼叫 panic! 巨集。使用 expect 呼叫 panic! 時的錯誤訊息會是傳遞給expect的參數，而不是像unwrap使用panic!預設的訊息。訊息看起來就會像這樣：

```
thread 'main' panicked at 'hello.txt 應該要存在此專案中: Os {
code: 2, kind: NotFound, message: "No such file or directory" }',
src/main.rs:5:10
```

  - 在正式環境等級的程式碼，大多數程式開發者會選擇 expect 而不是 unwrap，這樣能在出錯時提供更多資訊，告訴為何預期該動作永遠成功。這樣一來就算假設證明錯誤，都能夠在除錯時有足夠的資訊來理解。

錯誤處理

Result 與可復原的錯誤

- 傳播錯誤
  - 當函式實作呼叫的程式碼可能會失敗時，與其直接在該函式本身處理錯誤，可以回傳錯誤給呼叫此程式的程式碼，由它們決定如何處理。這稱之為傳播(propagating)錯誤並讓呼叫者可以有更多的控制權，因為比起程式碼當下的內容，回傳的錯誤可能提供更多資訊與邏輯以利處理。
  - 舉例來說，右上範例展示了一個從檔案讀取使用者名稱的函式。如果檔案不存在或無法讀取的話，此函式會回傳該錯誤給呼叫該函式的程式碼：

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

錯誤處理

Result 與可復原的錯誤

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

- 傳播錯誤
  - 此函式還能再更簡化，但要先繼續手動處理來進一步探討錯誤處理，最後會展示最精簡的方式。
  - 先看看此函式的回傳型別 Result<String, io::Error>。這代表此函式回傳的型別為 Result<T, E>，其中泛型型別 T 已經指明為實際型別 String，而泛型型別 E 則指明為實際型別 io::Error。
  - 如果函式正確無誤的話，程式碼會呼叫此函式並收到擁有 String 的 Ok 數值。如果程式遇到任何問題的話，呼叫此函式的程式碼就會獲得擁有包含相關問題發生資訊的 io::Error 實例的 Err 數值。選擇 io::Error 作為函式的回傳值是因為它正是 File::open 函式和 read_to_string 方法失敗時的回傳的錯誤型別。

錯誤處理

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

Result 與可復原的錯誤

- 傳播錯誤
  - 函式本體從呼叫 File::open 開始，然後使用 match 回傳 Result 數值，就和之前範例的 match 類似。
  - 如果 File::open 成功的話，變數 file 中的檔案控制代碼賦值給可變變數 username_file 並讓函式繼續執行下去。但在 Err 的情形時，與其呼叫 panic!，使用 return 關鍵字來讓函式提早回傳，並將 File::open 的錯誤值，也就是模式中的變數 e，作為此函式的錯誤值回傳給呼叫的程式碼。
  - 所以如果在 username_file 有拿到檔案控制代碼的話，接著函式就會在變數 username 建立新的 String 並對檔案控制代碼 username_file 呼叫 read_to_string 方法來讀取檔案內容至 username。read_to_string 也會回傳 Result 因為它也可能失敗，就算 File::open 是執行成功的。
  - 所以需要另一個 match 來處理該 Result，如果 read_to_string 成功的話，函式就是成功的，然後在 Ok 回傳 username 中該檔案的使用者名稱。如果 read_to_string 失敗的話，就像處理 File::open 的 match 一樣回傳錯誤值。**不過不需要顯式寫出 return，因為這是函式中的最後一個表達式**。

錯誤處理

Result 與可復原的錯誤

- 傳播錯誤
  - 呼叫此程式碼的程式就會需要處理包含使用者名稱的 Ok 數值以及包含 io::Error 的 Err 數值。這交給呼叫的程式碼來決定如何處理這些數值。
  - 舉例來說，如果呼叫此程式碼而獲得錯誤的話，它可能選擇呼叫 panic! 讓程式崩潰或者使用預設的使用者名稱從檔案以外的地方尋找該使用者。所以傳播所有成功或錯誤的資訊給呼叫者，讓它們能妥善處理。這樣傳播錯誤的模式是非常常見的，所以 Rust 提供了 ? 來簡化流程。

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

錯誤處理

Result 與可復原的錯誤

- 傳播錯誤的捷徑：? 運算子
  - 底下範例是另一個read_username_from_file的實作，擁有和之前範例一樣的效果，不過這次使用了?運算子：

```
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt")?;
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
```

  - 定義在 Result 數值後的 ? 運作方式幾乎與之前範例的 match 表達式處理 Result 的方式一樣。如果Result的數值是Ok的話，Ok 內的數值就會從此表達式回傳，然後程式就會繼續執行；如果數值是 Err 的話，Err 就會使用 return 關鍵字作為整個函式的回傳值回傳，讓錯誤數值可以傳遞給呼叫者的程式碼。

錯誤處理

Result 與可復原的錯誤

- 傳播錯誤的捷徑：? 運算子
  - 不過之前範例的 match 表達式做的事和 ? 運算子做的事還是有不同的地方：? 運算子呼叫所使用的錯誤數值會傳遞到 from 函式中，這是定義在標準函式庫的 From 特徵中，**用來將數值從一種型別轉換另一種型別。**
  - 當 ? 運算子呼叫 from 函式時，**接收到的錯誤型別會轉換成目前函式回傳值的錯誤型別**。這在當函式要回傳一個錯誤型別來代表所有函式可能的失敗是很有用的，即使可能會失敗的原因有很多種。
  - 舉例來說，可以將之前範例的函式 read_username_from_file 改成回傳一個自訂的錯誤型別叫做 OurError。如果有定義 impl From<io::Error> for OurError 能從 io::Error 建立一個 OurError 實例的話，那麼 read_username_from_file 本體中的 ? 運算就會呼叫 from 然後轉換錯誤型別，不必在函式那多加任何程式碼。

錯誤處理

Result 與可復原的錯誤

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt")?;
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
```

- 傳播錯誤的捷徑：? 運算子
  - 在之前範例中，在 File::open 的結尾中 ? 回傳 Ok 中的數值給變數 username_file。如果有錯誤發生時，? 運算子會提早回傳整個函式並將Err的數值傳給呼叫的程式碼。同理也適用在呼叫 read_to_string 結尾的 ?。
  - **? 運算子可以消除大量樣板程式碼並讓函式實作更簡單**。還可以再進一步將方法直接串接到 ? 後來簡化程式碼，如底下範例所示：

```rust
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username = String::new();

    File::open("hello.txt")?.read_to_string(&mut username)?;

    Ok(username)
}
```

將建立新 String 的變數 username 移到函式的開頭，這部分沒有任何改變。再來與建立變數 username_file 的地方不同的是，直接將 read_to_string 串接到 File::open("hello.txt")? 的結果後方。在 read_to_string 呼叫的結尾還是有 ?，然後還是在 File::open 和 read_to_string 成功沒有失敗時，回傳包含 username 的 Ok 數值。函式達成的效果仍然與之前範例相同。這只是一個比較不同但慣用的寫法。

錯誤處理

Result 與可復原的錯誤

- 傳播錯誤的捷徑：? 運算子
  - 說到此函式不同的寫法，底下範例展示了使用 **fs::read_to_string** 更短的寫法：

```
use std::fs;
use std::io;

fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}
```

  - 讀取檔案至字串中算是常見動作，所以標準函式庫提供了一個方便的函式 fs::read_to_string **來開啟檔案、建立新的String、讀取檔案內容、將內容放入該 String 並回傳它**。
  - 不過使用 fs::read_to_string 就沒有機會來解釋所有的錯誤處理，所以一開始才用比較長的寫法。

錯誤處理

Result 與可復原的錯誤

- ? 運算子可以用在哪裡？
  - **? 運算子只能用在有函式的回傳值相容於 ? 使用的值才行**。這是因為 ? 運算子會在函式中提早回傳數值，就像在之前範例那樣用 match 表達式提早回傳一樣。在之前範例中，match 使用的是 Result 數值，函式的回傳值必須是 Result 才能相容於此 return。
  - 看看在之前範例的 main 函式中的回傳值要是不相容用在 ? 的型別，如果使用 ? 運算子會發生什麼事：

```
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")?;
}
```

錯誤處理

Result 與可復原的錯誤

```rust
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")?;
}
```

- ? 運算子可以用在哪裡？
  - 此程式碼會開啟檔案，所以可能會失敗。**? 運算子會拿到 File::open 回傳的 Result 數值，但是此 main 函式的回傳值為() 而非 Result**。當編譯此程式碼時，會得到以下錯誤訊息：

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0277]: the `?` operator can only be used in a function that returns `Result` or `Option` (or
another type that implements `FromResidual`)
 --> src/main.rs:4:48
  |
3 | fn main() {
  | --------- this function should return `Result` or `Option` to accept `?`
4 |     let greeting_file = File::open("hello.txt")?;
  |                                                ^ cannot use the `?` operator in a function that
  returns `()`
  |
  = help: the trait `FromResidual<Result<Infallible, std::io::Error>>` is not implemented for `()`

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

錯誤處理

Result 與可復原的錯誤

```
$ cargo run
   Compiling error-handling v0.1.0 (file:///projects/error-handling)
error[E0277]: the `?` operator can only be used in a function that returns `Result` or `Option` (or
another type that implements `FromResidual`)
 --> src/main.rs:4:48
  |
3 | fn main() {
  | --------- this function should return `Result` or `Option` to accept `?`
4 |     let greeting_file = File::open("hello.txt")?;
  |                                                ^ cannot use the `?` operator in a function that
  returns `()`
  |
  = help: the trait `FromResidual<Result<Infallible, std::io::Error>>` is not implemented for `()`

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

- ? 運算子可以用在哪裡？
  - 此錯誤告訴只能在回傳型別為 Result 或 Option 或其他有實作FromResidual的型別的函式才能使用？運算子。
  - 要修正此錯誤的話，有兩種選擇。其中一種是如果沒有任何限制，可以將函式回傳值變更成與 ？ 運算子相容的型別。另一種則是依照可能的情境使用 match 或 Result<T, E> 其中一種方法來處理 Result<T, E>。
  - 錯誤訊息還提到了？也能用在 Option<T>的數值。就像？能用在Result一樣，只能在有回傳Option的函式中，對 Option 的值使用 ?。在 Option<T> 呼叫 ？ 的行為與在 Result<T, E> 上呼叫類似：如果數值為 None，None 就會在函式該處被提早回傳；如果數值為 Some，Some 中的值就會是表達式的結果數值，且程式會繼續執行下去。以下範例的函式會尋找給予文字的第一行中最後一個字元：

```
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

錯誤處理

Result 與可復原的錯誤

```
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

- ? 運算子可以用在哪裡？
  - 此函式會回傳 Option<char>，**因為它可能會在此真的找到字元或者可能根本沒有半個字存在**。此程式碼接受引數 text 字串切片，並呼叫它的 lines 方法，這會回傳一個走訪字串每一行的疊代器。因為此函式想要的是第一行，它對疊代器只呼叫 next 來取得疊代器的第一個數值。
  - 如果 text 是空字串的話，這裡 next 的呼叫就會回傳 None。這裡就可以使用 ? 來中斷 last_char_of_first_line 並回傳 None。如果 text 不是空字串的話，next 會用 Some 數值來回傳 text 的第一行字串切片。

錯誤處理

Result 與可復原的錯誤

```rust
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

- ? 運算子可以用在哪裡？
  - ? 會取出字串切片，然後可以對字串切片呼叫 chars 來取得它的疊代器。在意的是第一行的最後一個字元，所以呼叫 last 來取得疊代器的最後一個值。這也是個 Option 因為第一行可能是個空字串。如果 text 開頭就換行，但在下一行有字元的話，它可能就會是 "\nhi"。
  - 不過如果第一行真的有最後一個字元的話，它就會回傳 Some 變體。在這過程中的 ? 運算子能簡潔地表達此邏輯，並能只用一行就能實作出來。如果對 Option 無法使用 ? 運算子的話，使用更多方法呼叫或 match 表達式才能實作此邏輯。

錯誤處理

Result 與可復原的錯誤

```
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

- ? 運算子可以用在哪裡？
  - 請注意可以在有回傳 Result 的函式對 Result 的值使用 ? 運算子，可以在有回傳 Option 的函式對 Option 的值使用 ? 運算子，但無法混合使用。? 運算子無法自動轉換 Result 與 Option 之間的值。在這種狀況下會需要顯式表達，Result 的話有提供 ok 方法，Option 的話有提供 ok_or 方法。
  - 目前為止，所有使用過的 main 函式都是回傳()。main 是個特別的函式，因為它是可執行程式的進入點與出口點，而要讓程式可預期執行的話，它的回傳型別就得要有些限制。

錯誤處理

Result 與可復原的錯誤

```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let greeting_file = File::open("hello.txt")?;

    Ok(())
}
```

- ? 運算子可以用在哪裡？
  - 幸運的是 main 也可以回傳 Result<(), E>。右上範例取自之前範例，不過更改 main 的回傳型別為Result<(), Box<dyn Error>>，並在結尾的回傳數值加上 Ok(())。這樣的程式碼是能編譯的：
  - Box<dyn Error> 型別使用了特徵物件(trait object)，會在「允許不同型別數值的特徵物件」討論到。現在可以將 Box<dyn Error> 視為它是「任何種類的錯誤」。在有 Box<dyn Error> 錯誤型別的 main 函式中的 Result 使用 ? 是允許的，因為現在 Err 數值可以被提早回傳。
  - 盡管此 main 函式本來只會回傳錯誤型別 std::io::Error，但有了 Box<dyn Error> 的話，此簽名就能允許其他錯誤型別加入 main 本體中

錯誤處理

Result 與可復原的錯誤

```rust
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let greeting_file = File::open("hello.txt")?;

    Ok(())
}
```

- ? 運算子可以用在哪裡？
  - 當 main 函式回傳 Result<(), E> 時，如果 main 回傳 Ok(()) 的話，執行檔就會用 **0** 退出；如果 main 回傳 Err 數值的話，就會用**非零數值**退出。**用 C 語言寫的執行檔在退出時會回傳整數：程式成功退出的話會回傳整數 0，而程式退出錯誤的話則會回傳不是 0 的其他整數。而 Rust 執行檔也遵循相容這項規則。**
  - main 函式可以回傳任何有實作 std::process::Termination 特徵的型別，該特徵包含了一個函式 report 來回傳 ExitCode。可以查閱標準函式庫技術文件來了解如何對型別實作 Termination 特徵。
  - 現在已經討論了呼叫 panic! 與回傳 Result 的細節。現在回到**何時該使用何種辦法的主題上**。

錯誤處理

要 panic! 還是不要 panic!

- 該如何決定何時要呼叫 panic! 還是要回傳 Result 呢？當程式碼恐慌時，就沒有任何恢復的方式。可以在任何錯誤場合呼叫 panic!，無論是可能或不可能復原的情況。不過這樣就等於替呼叫程式碼的呼叫者做出決定，讓情況變成無法復原的錯誤了。
- 當選擇回傳 Result 數值，將決定權交給呼叫者的程式碼。**呼叫者可能會選擇符合當下場合的方式嘗試復原錯誤或者它可以選擇 Err 內的數值是不可恢復的，所以它就呼叫 panic! 讓原本可恢復的錯誤轉成不可恢復**。因此，當定義可能失敗的函式時預設回傳 Result 是不錯的選擇。
- 在像是範例、草稿與測試的情況下，程式碼恐慌會比回傳 Result 來得恰當。來探討為何比較好，然後再來討論編譯器無法辨別出不可能失敗，但身為人類卻可以的情況。最後會總結一些原則來決定何時在函式庫程式碼中恐慌。

錯誤處理

要 panic! 還是不要 panic!

- 範例、程式碼原型與測試
  - 當在寫解釋一些概念的範例時，寫出完善錯誤處理的範例，反而會讓範例變得較不清楚。在範例中，使用像是 unwrap 這樣會恐慌的方法可以被視為是一種要求使用者自行決定如何處理錯誤的表現，因為他們可以依據程式碼執行的方式來修改此方法。
  - 同樣地 unwrap 與 expect 方法也很適用在試做原型，可以在決定準備開始處理錯誤前使用它們。它們會留下清楚的痕跡，**當準備好要讓程式碼更穩固時，就能回來修改**。
  - 如果有方法在測試內失敗時，會希望整個測試都失敗，就算該方法不是要測試的功能。因為 panic! 會將測試標記為失敗，所以在此呼叫 unwrap 或 expect 是很正確的。

錯誤處理

要 panic! 還是不要 panic!

- 當知道的比編譯器還多的時候
  - 如果知道一些編譯器不知道的邏輯的話，直接在 Result 呼叫 unwrap 或 expect 來直接取得 Ok 的數值是很有用的。還是會有個 Result 數值需要做處理，呼叫的程式碼還是有機會失敗的，就算在特定場合中邏輯上是不可能的。
  - 如果能保證在親自審閱程式碼後，絕對不可能會有 Err 變體的話，那麼呼叫 unwrap 是完全可以接受的。而更好的話，用 expect 說明為何一定不會遇到 Err 變體。以下範例就是如此：

```
use std::net::IpAddr;

let home: IpAddr = "127.0.0.1"
    .parse()
    .expect("寫死的 IP 位址應該要有效");
```

錯誤處理

要 panic! 還是不要 panic!

```
use std::net::IpAddr;

let home: IpAddr = "127.0.0.1"
    .parse()
    .expect("寫死的 IP 位址應該要有效");
```

- 當知道的比編譯器還多的時候
  - 傳遞寫死的字串來建立 IpAddr 的實例。可以看出 127.0.0.1 是完全合理的 IP 位址，所以這邊可以直接 expect。不過使用寫死的合理字串並不會改變 parse 方法的回傳型別，還是會取得 Result 數值，編譯器仍然會要處理 Result 並認為 Err 變體是有可能發生的。
  - 因為編譯器並沒有聰明到可以看出此字串是個有效的 IP 位址。如果 IP 位址的字串是來自使用者輸入而非寫死進程式的話，它的確有可能會失敗，這時就得要認真處理 Result 了。註明該 IP 位址是寫死的能在未來想拿掉 expect 並改善錯誤處理時，幫助理解需要如何從其他來源處理 IP 位址。

錯誤處理

要 panic! 還是不要 panic!

- 錯誤處理的指導原則
  - **當程式碼可能會導致嚴重狀態的話，就建議讓程式恐慌。這裡的嚴重狀態是指一些假設、保證、協議或不可變性被打破時的狀態，像是當程式碼有無效的數值、互相矛盾的數值或缺少數值。**
  - 另外還加上以下情形：
    1. 該嚴重狀態並非預期會發生的，而不是像使用者輸入了錯誤格式這種偶而可能會發生的。
    2. 程式在此時需要避免這種嚴重狀態，而不是在每一步都處理此問題。
    3. 所使用的型別沒有適合的方式能夠處理此嚴重狀態。會在「定義狀態與行為成型別」段落用範例解釋指的是什麼。

錯誤處理

要 panic! 還是不要 panic!

- 錯誤處理的指導原則
  - 如果有人呼叫了程式碼卻傳遞了不合理的數值，如果可以的話最好的辦法是回傳個錯誤，這樣函式庫的使用者可以決定在該情況下該如何處理。不過要是繼續執行下去可能會造成危險或不安全的話，最好的辦法是呼叫 panic! 並警告使用函式庫的人的程式碼錯誤發生的位置，好讓在開發時就能修正。同樣地，panic! 也適合用於如果呼叫了無法掌控的外部程式碼，然後它回傳了無法修正的無效狀態。
  - 不過如果失敗是可預期的，回傳 Result 就會比呼叫 panic! 來得好。類似的例子有，語法分析器(parser)收到格式錯誤的資訊，或是 HTTP 請求回傳了一個狀態，告訴已經達到請求上限了。
  - 在這樣的案例，回傳 Result 代表失敗是預期有時會發生的，而且呼叫者必須決定如何處理。

錯誤處理

要 panic! 還是不要 panic!

- 錯誤處理的指導原則
  - 當程式碼可能會因為進行運算時輸入無效數值，而造成使用者安危的話，程式需要先驗證該數值，如果數值無效的話就要恐慌。這是基於安全原則，嘗試對無效資料做運算的話可能會導致程式碼產生漏洞。這也是標準函式庫在嘗試取得超出界限的記憶體存取會呼叫 panic! 的主要原因。
  - 嘗試取得不屬於當前資料結構的記憶體是常見的安全問題。函式通常都會訂下一些合約(contracts)，它們的行為只有在輸入資料符合特定要求時才帶有保障。當違反合約時恐慌是十分合理的，因為違反合約就代表這是呼叫者的錯誤，這不是程式碼該主動處理的錯誤。事實上，呼叫者也沒有任何合理的理由來復原這樣的錯誤。函式的合約應該要寫在函式的技術文件中解釋，尤其是違反時會恐慌的情況。

錯誤處理

要 panic! 還是不要 panic!

- 錯誤處理的指導原則
  - 然而要在函式寫一大堆錯誤檢查有時是很冗長且麻煩的。幸運的是可以利用 Rust 的型別系統(以及編譯器的型別檢查)來完成檢驗。如果函式用特定型別作為參數的話，就可以認定程式邏輯是編輯器已經確保拿到的數值是有效的。
  - 舉例來說，如果有一個型別而非 Option 的話，程式就會預期取得某個值而不是沒拿到值。程式就不必處理 Some 和 None 這兩個變體情形，它只會有一種情況並絕對會拿到數值。要是有人沒有傳遞任何值給函式會根本無法編譯，所以函式就不需要在執行時做檢查。另一個例子是使用非帶號整數像是 u32 來確保參數不會是負數。

錯誤處理

```rust
loop {
    // --省略--

    let guess: i32 = match guess.trim().parse() {
        Ok(num) => num,
        Err(_) => continue,
    };

    if guess < 1 || guess > 100 {
        println!("祕密數字介於 1 到 100 之間。");
        continue;
    }

    match guess.cmp(&secret_number) {
        // --省略--
}
```

要 panic! 還是不要 panic!

- 建立自訂型別來驗證
  - 來試著使用 Rust 的型別系統來進一步確保擁有有效數值，並建立自訂型別來驗證。例如猜謎遊戲的程式碼要使用者從 1 到 100 之間猜一個數字。在開始與祕密數字做比較之前，從未驗證使用者輸入的值，只驗證了它是否為正的。
  - 在這種情況帶來的後果還不算嚴重：還是會顯示「太大」或「太小」。但是可以改善這段來引導使用者輸入有效數值，並在使用者輸入時猜了超出範圍的數字或字母時呈現不同行為。
  - 可以將輸入的猜測分析改成 i32 而非 u32 來允許負數，並檢查數字是否在範圍內，如右上所示：
  - if 表達式檢查數值是否超出範圍，如果是的話就告訴使用者問題原因，並呼叫 continue 來進行下一次的猜測循環，要求再猜一次。在 if 表達式之後就能用已經知道範圍是在1到100的 guess 與祕密數字做比較。
  - 不過這並非理想解決方案：如果程式必定要求數值一定要是 1 到 100，而且有很多函式都有此需求的話，在每個函式都檢查就太囉唆了(而且可能會影響效能)。

錯誤處理

要 panic! 還是不要 panic!

- 建立自訂型別來驗證
  - 對此可以建立一個新的型別，並且建立一個驗證產生實例的函式，這樣就不必在每個地方都做驗證。這樣一來函式就可以安全地以這個新型別作為簽名，並放心地使用收到的數值。底下範例顯示了定義 Guess 型別的例子，它的 new 函式只會在接收值為 1 到 100 時才會建立 Guess 實例：

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("猜測數字必須介於 1 到 100 之間，你輸入的是 {}。", value);
        }

        Guess { value }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

錯誤處理

```rust
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("猜測數字必須介於 1 到 100 之間，你輸入的是 {}。", value);
        }

        Guess { value }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

要 panic! 還是不要 panic!

- 建立自訂型別來驗證
  - 首先定義了一個結構體叫做 Guess，其欄位叫做 value 並會持有 i32。這就是數字會被儲存的地方。
  - 接著實作一個 Guess 的關聯函式叫做 new 來建立 Guess 的值。new 函式定義的參數叫做 value 並擁有型別 i32，且最後會回傳 Guess。函式 new 本體中的程式碼會驗證 value 確保它位於 1 到 100 之間。
  - 如果 value 沒有通過驗證，呼叫 panic! 來警告呼叫此程式碼的開發者，可能有需要修正的程式錯誤，因為使用超出範圍的 value 來建立 Guess 違反了 Guess::new 的合約。
  - Guess::new 會恐慌的情況需要在公開的 API 技術文件中提及。會在之後如何寫出技術文件並在 API 技術文件中指出可能發生 panic! 的情形。如果 value 通過驗證的話，就建立一個新的 Guess 並將參數 value 賦值給 value 欄位，最後回傳 Guess。

錯誤處理

```rust
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("猜測數字必須介於 1 到 100 之間，你輸入的是 {}。", value);
        }

        Guess { value }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

要 panic! 還是不要 panic!

- 建立自訂型別來驗證
  - 接著實作了個方法叫做 value，它會借用 self 且沒有任何參數，並會回傳 i32。這種方法有時會被稱為 getter，因為它的目的是從它的欄位中取得一些資料並回傳它。
  - 此公開方法是必要的，因為 Guess 結構體中的 value 欄位是私有的。將 Guess 結構體的 value 欄位設為私有是很重要的，這樣就無法直接設置 value ，模組外的程式碼必須使用 Guess::new 函式來建立 Guess 的實例，因而確保 Guess 不可能會有沒有經過 Guess::new 函式驗證的 value。
  - 這樣當函式的參數或回傳值只能是數字 1 到 100 的話，它的簽名就能使用或回傳 Guess 而不是 i32，因此就不必在它的本體內做任何額外檢查。