


Uso del tipo Option para tratar la ausencia:

La biblioteca estándar de Rust proporciona una enumeración `Option<T>` que se usa cuando la ausencia de un valor es una posibilidad. `Option<T>` se utiliza mucho en el código de Rust. Resulta útil para trabajar con valores que pueden existir o que pueden estar vacíos.

En muchos otros lenguajes, esto se modela con `null` o `nil`, pero Rust no usa `null` fuera del código que interopera con otros lenguajes. Esto significa que Rust es explícito cuando un valor es opcional. Aunque en muchos lenguajes una función que toma `String` podría tomar `String` o un `null`, en Rust esa misma función solo puede tomar `Strings`. Si quiere modelar una cadena opcional en Rust, debe encapsularla explícitamente en un atributo `Option` tipo `Option<String>`.

`Option<T>` se manifiesta como una de estas dos variantes:

```
Rust
enum Option<T> {
    None,      // The value doesn't exist
    Some(T),   // The value exists
}
```

 Copiar

La parte `<T>` de la declaración de enumeración `Option<T>` indica que el tipo `T` es genérico y se asociará a la variante `Some` de la enumeración `Option`.

Tal como se ha explicado en las secciones anteriores, `None` y `Some` no son tipos, sino variantes del tipo `Option<T>`.

Esto significa, entre otras cosas, que las funciones no pueden tomar `Some` o `None` como argumentos, sino solo `Option<T>`.

En la unidad anterior, hemos mencionado que el intento de acceder al índice no existente de un vector haría que el programa emitiera una alerta panic, sin embargo, podría evitarlo mediante el método `Vec::get`, que devuelve un tipo `Option` en lugar de un error. Si el valor existe en un índice especificado, se encapsula en la variante `Option::Some(value)`. Si el índice está fuera de los límites, devolverá en cambio un valor `Option::None`.

```
Rust Copiar

let fruits = vec!["banana", "apple", "coconut", "orange", "strawberry"];

// pick the first item:
let first = fruits.get(0);
println!("{:?}", first);

// pick the third item:
let third = fruits.get(2);
println!("{:?}", third);

// pick the 99th item, which is non-existent:
let non_existent = fruits.get(99);
println!("{:?}", non_existent);
```

La salida muestra:

```
Resultados Copiar

Some("banana")
Some("coconut")
None
```

El mensaje impreso indica que los dos primeros intentos de acceder a los índices existentes en el vector `fruits` produjeron `Some("banana")` y `Some("coconut")`, pero el intento de capturar el elemento 99 devolvió un valor `None` (que no está asociado a ningún dato) en lugar de emitirse un panic.

En la práctica, debe decidirse cómo se comporta el programa en función del enum que se obtenga. Pero, ¿cómo se puede acceder a los datos dentro de una variante `Some(data)`?

Detección de patrones (Pattern matching):

En Rust, hay un operador de gran eficacia que se denomina `match`. Puede usarlo para controlar el flujo del programa mediante el aprovisionamiento de patrones. Cuando `match` encuentra un patrón coincidente, ejecuta el código que se proporciona con ese patrón.

Rust

Copy

```
let fruits = vec!["banana", "apple", "coconut", "orange", "strawberry"];
for &index in [0, 2, 99].iter() {
    match fruits.get(index) {
        Some(fruit_name) => println!("It's a delicious {}", fruit_name),
        None => println!("There is no fruit! :("),
    }
}
```

La salida muestra:

Output

Copy

```
It's a delicious banana!
It's a delicious coconut!
There is no fruit! :(
```

En el código anterior, se recorren en iteración los mismos índices del ejemplo previo (0, 2 y 99) y, luego, se usa cada uno de ellos para recuperar un valor del vector `fruits` mediante la expresión `fruits.get(index)`.

Dado que el vector `fruits` contiene elementos `&str`, sabemos que el resultado de esta expresión es de tipo `Option<&str>`. A continuación, se usará una expresión `match` con el valor `Option` y se definirá un curso de acción para cada una de sus variantes. Rust hace referencia a esas ramas como `match arms`, o secciones coincidentes, donde cada sección puede controlar un resultado posible para el valor coincidente.

La primera sección presenta una nueva variable, `fruit_name`. Esta variable coincide con cualquier valor dentro de un valor `Some`. El ámbito de `fruit_name` se limita a la expresión `match`, por lo que no tiene sentido declarar `fruit_name` antes de introducir esa opción en `match`.

Se puede refinar aún más la expresión `match` para que actúe de manera diferente, en función de los valores que existen dentro de una variante `Some`. Por ejemplo, puede hacer hincapié en el hecho de que los cocos son fabulosos ejecutando lo siguiente:

```
Rust Copiar
let fruits = vec!["banana", "apple", "coconut", "orange", "strawberry"];
for &index in [0, 2, 99].iter() {
    match fruits.get(index) {
        Some(&"coconut") => println!("Coconuts are awesome!!!"),
        Some(fruit_name) => println!("It's a delicious {}!", fruit_name),
        None => println!("There is no fruit! :("),
    }
}
```

❗ Nota

El primer patrón de la coincidencia es `Some(&"coconut")` (fíjese en el símbolo `&` anterior al literal de cadena). Esto se debe a que `fruits.get(index)` devuelve `Option<&str>` o una opción de una referencia a un segmento de cadena. Al retirar `&` del patrón, se está intentando buscar coincidencias con `Option<str>` (un segmento de cadena opcional, *no* una referencia opcional a un segmento de cadena). Como no hemos analizado las referencias, puede ser que en estos momentos no acabe de verlo claro. Por ahora, recuerde que `&` se asegura de que los tipos se alinean correctamente.

La salida muestra:

```
Resultados Copiar
It's a delicious banana!
Coconuts are awesome!!!
There is no fruit! :(
```

Tenga en cuenta que, cuando el valor de la cadena es "coconut", se busca una coincidencia con la primera sección y, luego, se usa esta para determinar el flujo de ejecución.

Siempre que use la expresión `match`, tenga en cuenta las siguientes reglas:

- Las secciones `match` se evalúan de arriba hacia abajo. Los casos específicos se deben definir antes que los casos genéricos o nunca se buscará una coincidencia para ellos ni se evaluarán.
- Las secciones `match` deben cubrir todos los valores posibles que pueda tener el tipo de entrada. Si intenta buscar coincidencias con una lista de patrones no exhaustiva, recibirá un error de compilador.

La expresión `if let`

Rust ofrece una manera cómoda de probar si un valor se ajusta a un solo patrón.

En el ejemplo siguiente, la entrada a `match` es un valor `Option<u8>`. La expresión `match` solo debe ejecutar código si ese valor de entrada es `7`.


```
Rust Copy  
  
let a_number: Option<u8> = Some(7);  
match a_number {  
    Some(7) => println!("That's my lucky number!"),  
    _ => {},  
}
```

En este caso, nos gustaría ignorar la variante `None` y todos los valores de `Some<u8>` que no coincidan con `Some(7)`.

Los patrones comodín son útiles para este tipo de situación. Puede agregar el patrón de carácter comodín (`_` underscore) a todos los demás patrones que coincidan con *cualquier otra cosa*; este patrón se usa para satisfacer las demandas del compilador de agotar las secciones de coincidencia.

Para condensar este código, puede usar una expresión `if let`:

Rust

 Copy

```
let a_number: Option<u8> = Some(7);
if let Some(7) = a_number {
    println!("That's my lucky number!");
}
```

Un operador **if let** compara un patrón con una expresión.

Si la expresión coincide con el patrón, se ejecuta el bloque **if**. Lo bueno de la expresión **if let** es que no se necesita todo el código reutilizable de una expresión **match** cuando solo interesa un patrón con el que buscar coincidencias.

Uso de unwrap y expect:

Puede intentar acceder al valor interno de un tipo **Option** directamente mediante el **método unwrap**. Sin embargo, tenga cuidado, ya que este método emitirá un **panic** si la variante es **None**.

Por ejemplo:

Rust


 Copiar

```
let gift = Some("candy");
assert_eq!(gift.unwrap(), "candy");

let empty_gift: Option<&str> = None;
assert_eq!(empty_gift.unwrap(), "candy"); // This will panic!
```

En este caso, el código emitiría una alerta de pánico con el siguiente resultado:


Resultados

 Copiar

```
thread 'main' panicked at 'called `Option::unwrap()` on a `None` value', src/main.rs:6:27
```

El método **expect** hace lo mismo que **unwrap**, pero emite un **panic** personalizado pudiendo pasárselo como argumento:

Rust


 Copiar

```
let a = Some("value");
assert_eq!(a.expect("fruits are healthy"), "value");

let b: Option<&str> = None;
b.expect("fruits are healthy"); // panics with `fruits are healthy`
```

La salida muestra:

Resultados


 Copiar

```
thread 'main' panicked at 'fruits are healthy', src/main.rs:6:7
```

Como estas funciones pueden emitir panics, no se recomienda usarlas. Considere mejor uno de los siguientes enfoques:

- Use la coincidencia de patrones y administre el caso `None` explícitamente.
- Llame a métodos similares que no emiten panics, como `unwrap_or`, que devuelve un valor predeterminado si la variante es `None` o el valor interno si la variante es `Some(value)`.

Rust

 Copiar

```
assert_eq!(Some("dog").unwrap_or("cat"), "dog");
assert_eq!(None.unwrap_or("cat"), "cat");
```