

Uso de variantes de enumeración para datos compuestos:

Las enumeraciones (enums) son tipos que pueden ser una de varias variantes. Lo que Rust denomina enumeraciones se conocen habitualmente como [tipos de datos algebraicos](#). Lo importante es que cada variante de enumeración puede tener datos asociados.

Usamos la palabra clave **enum** para crear un enum, que puede tener cualquier combinación de las variantes de enumeración. Los enums, al igual que las estructuras, pueden tener campos con nombres, pero también los pueden tener sin nombres, o bien no tener ningún campo. Al igual que los tipos de estructura, los enums también se escriben en mayúsculas.

Definición de una enumeración:

En el ejemplo siguiente, se define una enumeración para clasificar un evento web. Cada variante de la enumeración es independiente y almacena diferentes cantidades y tipos de valores.

Rust

 Copiar

```
enum WebEvent {  
    // An enum variant can be like a unit struct without fields or data types  
    WELoad,  
    // An enum variant can be like a tuple struct with data types but no named fields  
    WEKeys(String, char),  
    // An enum variant can be like a classic struct with named fields and their data types  
    WEClick { x: i64, y: i64 }  
}
```

La enumeración de nuestro ejemplo tiene tres variantes de tipos diferentes:

- **WELoad** no tiene ningún tipo de dato o datos asociados.
- **WEKeys** tiene dos campos, con tipos de datos String y char.
- **WEClick** incluye una estructura anónima con campos con nombre x e y, y sus tipos de datos (i64).

Definimos un **enum** con variantes parecidas a la forma en que definimos diferentes clases de tipos de estructura. Todas las variantes se agrupan en el mismo tipo de enumeración WebEvent. Cada variante de la enumeración no es su propio tipo. Cualquier función que use una variante de la enumeración WebEvent debe aceptar todas las

variantes de esta. No podemos tener una función que acepte solo la variante WEClick, pero no las demás.

Definición de una enumeración con estructuras

Una manera de evitar los requisitos de variante de la enumeración es definir una estructura independiente para cada variante de esta.

Después, cada variante de la enumeración usa la estructura correspondiente. La estructura contiene los mismos datos que tenía la variante de enumeración correspondiente. Este estilo de definición nos permite hacer referencia a cada variante lógica por sí misma.

En el código siguiente se muestra cómo utilizar este estilo de definición alternativo. Las estructuras se definen para contener los datos. Las variantes de la enumeración se definen para hacer referencia a las estructuras.

Rust

 Copiar

```
// Define a tuple struct
struct KeyPress(String, char);

// Define a classic struct
struct MouseButton { x: i64, y: i64 }

// Redefine the enum variants to use the data from the new structs
// Update the page Load variant to have the boolean type
enum WebEvent { WELoad(bool), WEClick(MouseButton), WEKeys(KeyPress) }
```

Instanciando un enum:

Ahora vamos a agregar código para crear instancias de nuestras variantes de enumeración. Para cada variante, usamos la palabra clave `let` a fin de realizar la asignación. Para acceder a la variante específica en la definición de enumeración, usamos la sintaxis `<enum>::<variant>` con dos puntos dobles `::`.

Variante simple: WELoad(bool)

La primera variante de la enumeración `WebEvent` tiene un único valor booleano, `WELoad(bool)`. Creamos una instancia de esta variante de forma parecida a como hemos trabajado con los valores booleanos de la unidad anterior:

Rust


 Copiar

```
let we_load = WebEvent::WELoad(true);
```

Variante de estructura: WEClick(MouseClick)

La segunda variante incluye una (classic struct) WEClick(MouseClick). La estructura tiene dos campos con nombre x e y, y ambos campos tienen el tipo de datos i64. Para crear esta variante, en primer lugar creamos una instancia de la estructura. Después, pasamos la estructura como argumento en la llamada para crear una instancia de la variante.

Rust

 Copiar


```
// Instantiate a MouseClick struct and bind the coordinate values
let click = MouseClick { x: 100, y: 250 };

// Set the WEClick variant to use the data in the click struct
let we_click = WebEvent::WEClick(click);
```

Variante de tupla: WEKeys(KeyPress)

La última variante incluye una tupla WEKeys(KeyPress). La tupla tiene dos campos que usan los tipos de datos String y char. Para crear esta variante, primero creamos una instancia de la tupla. Después, pasamos la tupla como argumento en la llamada para crear una instancia de la variante.

Rust

 Copiar

```
// Instantiate a KeyPress tuple and bind the key values
let keys = KeyPress(String::from("Ctrl+"), 'N');


// Set the WEKeys variant to use the data in the keys tuple
let we_key = WebEvent::WEKeys(keys);
```

Observe que usamos la sintaxis `String::from("<value>")` en este fragmento de código. Esta sintaxis crea un valor de tipo String llamando al método `from` de Rust. El método espera un argumento de entrada de datos entre comillas dobles.

Ejemplo de enumeraciones

Este es el código final para crear una instancia de las variantes de enumeración:

Rust

 Copiar

```
// Define a tuple struct
#[derive(Debug)]
struct KeyPress(String, char);

// Define a classic struct
#[derive(Debug)]
struct MouseButton { x: i64, y: i64 }

// Define the WebEvent enum variants to use the data from the structs
// and a boolean type for the page Load variant
#[derive(Debug)]
enum WebEvent { WELoad(bool), WEClick(MouseButton), WEKeys(KeyPress) }

// Instantiate a MouseButton struct and bind the coordinate values
let click = MouseButton { x: 100, y: 250 };
println!("Mouse click location: {}, {}", click.x, click.y);

// Instantiate a KeyPress tuple and bind the key values
let keys = KeyPress(String::from("Ctrl+"), 'N');
println!("Keys pressed: {}", keys.0, keys.1);

// Instantiate WebEvent enum variants
// Set the boolean page Load value to true
let we_load = WebEvent::WELoad(true);
// Set the WEClick variant to use the data in the click struct
let we_click = WebEvent::WEClick(click);
// Set the WEKeys variant to use the data in the keys tuple
let we_key = WebEvent::WEKeys(keys);

// Print the values in the WebEvent enum variants
// Use the {:#?} syntax to display the enum structure and data in a readable form
println!("WebEvent enum structure: \n\n {:#?} \n\n {:#?} \n\n {:#?}", we_load, we_click, we_key);
```

Instrucciones de depuración

En el ejemplo anterior, busque la siguiente instrucción de código. Esta instrucción se usa en varios lugares del código.

Rust

 Copiar

```
// Set the Debug flag so we can check the data in the output
#[derive(Debug)]
```

La sintaxis `#[derive(Debug)]` nos permite ver determinados valores durante la ejecución del código que, de lo contrario, no son visibles en la salida estándar. Para ver los datos de depuración con la macro `println!`, usamos la sintaxis `{:#?}` a fin de dar formato a los datos de una manera legible.