Tipos personalizados

Os tipos de dados personalizados do Rust são formados principalmente por meio de duas palavras-chave:

- `struct` : definir uma estrutura
- `enum` : definir uma enumeração

Constantes também podem ser criadas por meio das palavras-chave `const` e `static` .

Estruturas

Existem três tipos de estruturas ("structs") que podem ser criadas usando a `struct` palavra-chave:

- Estruturas de tuplas, que são, basicamente, chamadas de tuplas.
- As estruturas C clássicas
- Estruturas unitárias, que não possuem campos, são úteis para genéricos.

```rust
// An attribute to hide warnings for unused code.
#![allow(dead_code)]

#[derive(Debug)]
struct Person {
    name: String,
    age: u8,
}

// A unit struct
struct Unit;

// A tuple struct
struct Pair(i32, f32);

// A struct with two fields
struct Point {
    x: f32,
    y: f32,
}

// Structs can be reused as fields of another struct
struct Rectangle {
    // A rectangle can be specified by where the top left and bottom right
    // corners are in space.
    top_left: Point,
    bottom_right: Point,
}

fn main() {
    // Create struct with field init shorthand
    let name = String::from("Peter");
    let age = 27;
    let peter = Person { name, age };

    // Print debug struct
    println!("{:?}", peter);

    // Instantiate a `Point`
    let point: Point = Point { x: 5.2, y: 0.4 };
    let another_point: Point = Point { x: 10.3, y: 0.2 };

    // Access the fields of the point
    println!("point coordinates: ({}, {})", point.x, point.y);

    // Make a new point by using struct update syntax to use the fields of o
    // other one
    let bottom_right = Point { x: 10.3, ..another_point };

    // `bottom_right.y` will be the same as `another_point.y` because we use
    // from `another_point`
    println!("second point: ({}, {})", bottom_right.x, bottom_right.y);

    // Destructure the point using a `let` binding
    let Point { x: left_edge, y: top_edge } = point;

    let _rectangle = Rectangle {
        // struct instantiation is an expression too
```

```
59          top_left: Point { x: left_edge, y: top_edge },
60          bottom_right: bottom_right,
61      };
62
63      // Instantiate a unit struct
64      let _unit = Unit;
65
66      // Instantiate a tuple struct
67      let pair = Pair(1, 0.1);
68
69      // Access the fields of a tuple struct
70      println!("pair contains {:?} and {:?}", pair.0, pair.1);
71
72      // Destructure a tuple struct
73      let Pair(integer, decimal) = pair;
74
75      println!("pair contains {:?} and {:?}", integer, decimal);
76  }
```

Atividade

1. Adicione uma função `rect_area` que calcule a área de a `Rectangle` (tente usar a desestruturação aninhada).
2. Adicione uma função `square` que receba a `Point` e a `f32` como argumentos e retorne a `Rectangle` com seu canto superior esquerdo no ponto e uma largura e altura correspondentes a `f32` .

Veja também

`attributes` , identificadores brutos e desestruturação

Enumerações

A `enum` palavra-chave permite a criação de um tipo que pode ser uma de algumas variantes diferentes. Qualquer variante válida como a `struct` também é válida em an `enum` .

```
// Create an `enum` to classify a web event. Note how both
// names and type information together specify the variant:
// `PageLoad != PageUnload` and `KeyPress(char) != Paste(String)`.
// Each is different and independent.
enum WebEvent {
    // An `enum` variant may either be `unit-like`,
    PageLoad,
    PageUnload,
    // like tuple structs,
    KeyPress(char),
    Paste(String),
    // or c-like structures.
    Click { x: i64, y: i64 },
}

// A function which takes a `WebEvent` enum as an argument and
// returns nothing.
fn inspect(event: WebEvent) {
    match event {
        WebEvent::PageLoad => println!("page loaded"),
        WebEvent::PageUnload => println!("page unloaded"),
        // Destructure `c` from inside the `enum` variant.
        WebEvent::KeyPress(c) => println!("pressed '{}'.", c),
        WebEvent::Paste(s) => println!("pasted \"{}\".", s),
        // Destructure `Click` into `x` and `y`.
        WebEvent::Click { x, y } => {
            println!("clicked at x={}, y={}.", x, y);
        },
    }
}

fn main() {
    let pressed = WebEvent::KeyPress('x');
    // `to_owned()` creates an owned `String` from a string slice.
    let pasted  = WebEvent::Paste("my text".to_owned());
    let click   = WebEvent::Click { x: 20, y: 80 };
    let load    = WebEvent::PageLoad;
    let unload  = WebEvent::PageUnload;

    inspect(pressed);
    inspect(pasted);
    inspect(click);
    inspect(load);
    inspect(unload);
}

```

Aliases de tipo

Se você usar um alias de tipo, poderá se referir a cada variante de enumeração por meio do seu alias. Isso pode ser útil se o nome da enumeração for muito longo ou genérico e você quiser renomeá-lo.

```
enum VeryVerboseEnumOfThingsToDoWithNumbers {
    Add,
    Subtract,
}

// Creates a type alias
type Operations = VeryVerboseEnumOfThingsToDoWithNumbers;

fn main() {
    // We can refer to each variant via its alias, not its long and inconver
    // name.
    let x = Operations::Add;
}
```

O lugar mais comum onde você verá isso é em `impl` blocos que usam o `Self` alias.

```
enum VeryVerboseEnumOfThingsToDoWithNumbers {
    Add,
    Subtract,
}

impl VeryVerboseEnumOfThingsToDoWithNumbers {
    fn run(&self, x: i32, y: i32) -> i32 {
        match self {
            Self::Add => x + y,
            Self::Subtract => x - y,
        }
    }
}
```

Para saber mais sobre enumerações e aliases de tipo, você pode ler o relatório de estabilização de quando esse recurso foi estabilizado no Rust.

Veja também:

`match`, `fn`, e `String`, "Variantes de enumeração de alias de tipo" RFC

usar

A `use` declaração pode ser usada para que o escopo manual não seja necessário:

```
// An attribute to hide warnings for unused code.
#![allow(dead_code)]

enum Stage {
    Beginner,
    Advanced,
}

enum Role {
    Student,
    Teacher,
}

fn main() {
    // Explicitly `use` each name so they are available without
    // manual scoping.
    use crate::Stage::{Beginner, Advanced};
    // Automatically `use` each name inside `Role`.
    use crate::Role::*;

    // Equivalent to `Stage::Beginner`.
    let stage = Beginner;
    // Equivalent to `Role::Student`.
    let role = Student;

    match stage {
        // Note the lack of scoping because of the explicit `use` above.
        Beginner => println!("Beginners are starting their learning journey!
        Advanced => println!("Advanced learners are mastering their subjects
    }

    match role {
        // Note again the lack of scoping.
        Student => println!("Students are acquiring knowledge!"),
        Teacher => println!("Teachers are spreading knowledge!"),
    }
}
```

Veja também:

`match` e `use`

tipo C

`enum` também podem ser usados como enumerações do tipo C.

```rust
// An attribute to hide warnings for unused code.
#![allow(dead_code)]

// enum with implicit discriminator (starts at 0)
enum Number {
    Zero,
    One,
    Two,
}

// enum with explicit discriminator
enum Color {
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff,
}

fn main() {
    // `enums` can be cast as integers.
    println!("zero is {}", Number::Zero as i32);
    println!("one is {}", Number::One as i32);

    println!("roses are #{:06x}", Color::Red as i32);
    println!("violets are #{:06x}", Color::Blue as i32);
}
```

Veja também:

fundição

constantes

Rust possui dois tipos diferentes de constantes que podem ser declaradas em qualquer escopo, incluindo global. Ambas exigem anotação de tipo explícita:

- `const` : Um valor imutável (o caso comum).
- `static` : Uma variável possivelmente mutável com `'static` tempo de vida. O tempo de vida estático é inferido e não precisa ser especificado. Acessar ou modificar uma variável estática mutável é `unsafe` .

```
// Globals are declared outside all other scopes.
static LANGUAGE: &str = "Rust";
const THRESHOLD: i32 = 10;

fn is_big(n: i32) -> bool {
    // Access constant in some function
    n > THRESHOLD
}

fn main() {
    let n = 16;

    // Access constant in the main thread
    println!("This is {}", LANGUAGE);
    println!("The threshold is {}", THRESHOLD);
    println!("{} is {}", n, if is_big(n) { "big" } else { "small" });

    // Error! Cannot modify a `const`.
    THRESHOLD = 5;
    // FIXME ^ Comment out this line
}
```

Veja também:

O `const` / `static` RFC , `'static` tempo de vida