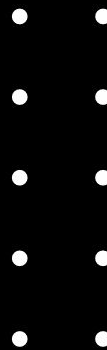


Manejo de paquetes



Oscar Julian Cardenas



NACION
CRYPTO

By



DECRY

Cargo

Cargo es el sistema de construcción y administración de paquetes para proyectos Rust. Su función principal es simplificar el proceso de desarrollo, compilación y distribución de proyectos Rust.

1	cargo build Compiles the current project.
2	cargo check Analyzes the current project and report errors, but don't build object files.
3	cargo run Builds and executes <code>src/main.rs</code> .
4	cargo clean Removes the target directory.
5	cargo update Updates dependencies listed in <code>Cargo.lock</code> .
6	cargo new Creates a new cargo project.

Modules

Los módulos facilitan la creación de abstracciones y ayudan a gestionar la complejidad de los proyectos. Puedes agrupar funciones, estructuras, enums y otros elementos relacionados en módulos para mejorar la legibilidad y la mantenibilidad del código.

Visibilidad

```
// A module named `my_mod`
mod my_mod {
    // Items in modules default to private visibility.
    fn private_function() {
        println!("called `my_mod::private_function()`");
    }

    // Use the `pub` modifier to override default visibility.
    pub fn function() {
        println!("called `my_mod::function()`");
    }

    // Items can access other items in the same module,
    // even when private.
    pub fn indirect_access() {
        print!("called `my_mod::indirect_access()`, that\n> ");
        private_function();
    }
}
```

Visibilidad de los struct

Los Structs tienen un nivel extra de visibilidad con sus campos. La visibilidad por defecto es `private`, y puede ser anulada con el modificador `pub`. Esta visibilidad sólo importa cuando se accede a un struct desde fuera del módulo donde está definido, y tiene el objetivo de ocultar información (encapsulación).

```
mod my {
  // A public struct with a public field of generic type `T`
  pub struct OpenBox<T> {
    pub contents: T,
  }

  // A public struct with a private field of generic type `T`
  pub struct ClosedBox<T> {
    contents: T,
  }

  impl<T> ClosedBox<T> {
    // A public constructor method
    pub fn new(contents: T) -> ClosedBox<T> {
      ClosedBox {
        contents: contents,
      }
    }
  }
}
```

Use declaration

La declaración de uso se puede utilizar para vincular una ruta completa a un nuevo nombre, para facilitar el acceso. En otras palabras, permite importar funciones que se encuentren dentro de un módulo sin tener que volverlas a declarar

```
// Bind the `deeply::nested::function` path to `other_function`.
use deeply::nested::function as other_function;

fn function() {
    println!("called `function()`");
}

mod deeply {
    pub mod nested {
        pub fn function() {
            println!("called `deeply::nested::function()`");
        }
    }
}
```

Self

```
struct MiEstructura {
    dato: i32,
}

impl MiEstructura {
    fn obtener_dato(&self) -> i32 {
        self.dato
    }
}

fn main() {
    let instancia = MiEstructura { dato: 42 };
    println!("Dato: {}", instancia.obtener_dato());
}
```

Super

```
mod modulo_padre {
    pub fn funcion_padre() {
        println!("¡Hola desde el módulo padre!");
    }
}

mod modulo_hijo {
    use super::funcion_padre;

    pub fn funcion_hijo() {
        println!("¡Hola desde el módulo hijo!");
        funcion_padre();
    }
}

fn main() {
    modulo_hijo::funcion_hijo();
}
```

Creates

Crear una Librería

En `rary.rs`

```
pub fn public_function() {
    println!("called rary's `public_function()`");
}

fn private_function() {
    println!("called rary's `private_function()`");
}

pub fn indirect_access() {
    print!("called rary's `indirect_access()`, that\n> ");

    private_function();
}
```

```
$ rustc --crate-type=lib rary.rs
$ ls lib*
library.rlib
```

Usar una librería



```
// extern crate rary;  
  
fn main() {  
    rary::public_function();  
  
    // Error! `private_function` is private  
    // rary::private_function();  
  
    rary::indirect_access();  
}
```

```
# Where library.rlib is the path to the compiled library, assumed that it's  
# in the same directory here:  
$ rustc executable.rs --extern rary=library.rlib && ./executable  
called rary's `public_function()`  
called rary's `indirect_access()`, that  
> called rary's `private_function()`
```


block-buffer

```
use blockcuffer::BlockBuffer;

fn main() {
    let buffer = BlockBuffer::new(1024, 16);

    // Insert some data into the buffer.

    for i in 0..1024 {
        buffer[i] = i as u8;
    }

    // Read some data from the buffer.

    for i in 0..1024 {
        assert_eq!(buffer[i], i as u8);
    }

    // Clear the buffer.

    buffer.clear();

    // Check if the buffer is empty.

    assert_eq!(buffer.len(), 0);
}
```

Serde JSON

```

use serde_json_experimental;

#[derive(Serialize, Deserialize)]
struct List {
    items: Vec<i32>,
}

fn main() {
    let data = r#"[1, 2, 3, 4, 5]"#;

    let list: List = serde_json_experimental::from_str(data).unwrap();

    println!("La lista contiene los siguientes elementos:");
    for item in list.items {
        println!("{}", item);
    }
}

```

BIBLIOGRAFIA:

- <https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html>
- https://www.tutorialspoint.com/rust/rust_package_manager.htm
- Cargo book:
<https://doc.rust-lang.org/cargo/index.html>
- <https://crates.io/>