EBook Gratis

APRENDIZAJE Rust

Free unaffiliated eBook created from
Stack Overflow contributors.

#rust

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: rust

It is an unofficial and free Rust ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Rust.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Rust

Observaciones

Rust es un lenguaje de programación de sistemas diseñado para seguridad, velocidad y concurrencia. Rust tiene numerosas funciones de tiempo de compilación y controles de seguridad para evitar las carreras de datos y los errores comunes, todo con una sobrecarga mínima de tiempo de ejecución cero.

Versiones

Estable

Versión	Fecha de lanzamiento
1.17.0	2017-04-27
1.16.0	2017-03-16
1.15.1	2017-02-09
1.15.0	2017-02-02
1.14.0	2016-12-22
1.13.0	2016-11-10
1.12.1	2016-10-20
1.12.0	2016-09-30
1.11.0	2016-08-18
1.10.0	2016-07-07
1.9.0	2016-05-26
1.8.0	2016-04-14
1.7.0	2016-03-03
1.6.0	2016-01-21
1.5.0	2015-12-10
1.4.0	2015-10-29

Versión	Fecha de lanzamiento
1.3.0	2015-09-17
1.2.0	2015-08-07
1.1.0	2015-06-25
1.0.0	2015-05-15

Beta

Versión	Fecha de lanzamiento prevista
1.18.0	2017-06-08

Examples

Uso avanzado de println!

`println!` (¡y su hermano, `print!`) proporciona un mecanismo conveniente para producir e imprimir texto que contiene datos dinámicos, similar a la familia de funciones `printf` encuentran en muchos otros idiomas. Su primer argumento es una *cadena de formato* , que dicta cómo los otros argumentos deben imprimirse como texto. La cadena de formato puede contener marcadores de posición (incluidos en `{}`) para especificar que debe producirse una sustitución:

```
// No substitution -- the simplest kind of format string
println!("Hello World");
// Output: Hello World

// The first {} is substituted with a textual representation of
// the first argument following the format string. The second {}
// is substituted with the second argument, and so on.
println!("{} {} {}", "Hello", true, 42);
// Output: Hello true 42
```

En este punto, es posible que se pregunte: ¡Cómo se `println!` ¿Sabes para imprimir el valor booleano `true` como la cadena "verdadero"? `{}` es realmente una instrucción para el formateador de que el valor se debe convertir en texto usando el rasgo de `Display` . Este rasgo se implementa para la mayoría de los tipos de Rust primitivos (cadenas, números, booleanos, etc.) y está destinado a "salida orientada al usuario". Por lo tanto, el número 42 se imprimirá en decimal como 42, y no, digamos, en binario, que es cómo se almacena internamente.

¿Cómo imprimimos tipos, entonces, que *no* implementan `Display` , siendo los ejemplos Slices (`[i32]`), vectores (`Vec<i32>`) u opciones (`Option<&str>`)? No hay una clara representación textual de éstos (p. Ej., Una que podría insertar de forma trivial en una oración). Para facilitar la impresión de dichos valores, Rust también tiene el rasgo de `Debug` y el correspondiente marcador de posición `{:?}` . De la documentación: "La `Debug` debe formatear la salida en un contexto de depuración

orientado hacia el programador". Veamos algunos ejemplos:

```
println!("{:?}", vec!["a", "b", "c"]);
// Output: ["a", "b", "c"]

println!("{:?}", Some("fantastic"));
// Output: Some("fantastic")

println!("{:?}", "Hello");
// Output: "Hello"
// Notice the quotation marks around "Hello" that indicate
// that a string was printed.
```

`Debug` también tiene un mecanismo de impresión bonito incorporado, que puede habilitar usando el modificador `#` después de los dos puntos:

```
println!("{:#?}", vec![Some("Hello"), None, Some("World")]);
// Output: [
//     Some(
//         "Hello"
//     ),
//     None,
//     Some(
//         "World"
//     )
// ]
```

Las cadenas de formato le permiten expresar sustituciones bastante complejas :

```
// You can specify the position of arguments using numerical indexes.
println!("{1} {0}", "World", "Hello");
// Output: Hello World

// You can use named arguments with format
println!("{greeting} {who}!", greeting="Hello", who="World");
// Output: Hello World

// You can mix Debug and Display prints:
println!("{greeting} {1:?}, {0}", "and welcome", Some(42), greeting="Hello");
// Output: Hello Some(42), and welcome
```

`println!` y los amigos también te avisarán si estás tratando de hacer algo que no funcionará, en lugar de estrellarte en el tiempo de ejecución:

```
// This does not compile, since we don't use the second argument.
println!("{}", "Hello World", "ignored");

// This does not compile, since we don't give the second argument.
println!("{} {}", "Hello");

// This does not compile, since Option type does not implement Display
println!("{}", Some(42));
```

En su núcleo, las macros de impresión Rust son simplemente envoltorios alrededor del `format!` macro, que permite construir una cadena uniendo representaciones textuales de diferentes

valores de datos. Por lo tanto, para todos los ejemplos anteriores, puede sustituir `println!` para el `format!` para almacenar la cadena formateada en lugar de imprimirla:

```
let x: String = format!("{} {}", "Hello", 42);
assert_eq!(x, "Hello 42");
```

Salida de consola sin macros

```
// use Write trait that contains write() function
use std::io::Write;

fn main() {
    std::io::stdout().write(b"Hello, world!\n").unwrap();
}
```

- El rasgo `std::io::Write` está diseñado para objetos que aceptan flujos de bytes. En este caso, se adquiere un manejador de salida estándar con `std::io::stdout()` .
- `Write::write()` acepta un segmento de bytes (`&[u8]`), que se crea con un literal de cadena de bytes (`b"<string>"`). `Write::write()` devuelve un `Result<usize, IoError>` , que contiene el número de bytes escritos (en caso de éxito) o un valor de error (en caso de error).
- La llamada a `Result::unwrap()` indica que se espera que la llamada sea exitosa (`Result<usize, IoError> -> usize`), y el valor se descarta.

Ejemplo minimo

Para escribir el programa tradicional Hello World en Rust, cree un archivo de texto llamado `hello.rs` contenga el siguiente código fuente:

```
fn main() {
    println!("Hello World!");
}
```

Esto define una nueva función llamada `main` , que no toma parámetros y no devuelve datos. Aquí es donde su programa comienza a ejecutarse cuando se ejecuta. En su interior, tienes un `println!` , que es una macro que imprime texto en la consola.

Para generar una aplicación binaria, invoque el compilador Rust pasándole el nombre del archivo fuente:

```
$ rustc hello.rs
```

El ejecutable resultante tendrá el mismo nombre que el módulo fuente principal, por lo que para ejecutar el programa en un sistema Linux o MacOS, ejecute:

```
$ ./hello
Hello World!
```

En un sistema Windows, ejecute:

```
C:\Rust> hello.exe
Hello World!
```

Empezando

Instalación

Antes de poder hacer algo con el lenguaje de programación Rust, deberá adquirirlo, ya sea para Windows o para usar su terminal en *sistemas similares a* Unix, donde `$` simboliza la entrada en el terminal:

```
$ curl https://sh.rustup.rs -sSf | sh
```

Esto recuperará los archivos necesarios y configurará la última versión de Rust para usted, sin importar en qué sistema se encuentre. Para más información, vea la página del proyecto .

Nota: Algunas distribuciones de Linux (p. Ej., Arch Linux) proporcionan el proceso de `rustup` como un paquete, que puede instalarse en su lugar. Y aunque muchos sistemas similares a Unix proporcionan `rustc` y `cargo` como paquetes separados, todavía se recomienda usar `rustup` ya que hace que sea mucho más fácil administrar múltiples canales de lanzamiento y realizar compilación cruzada.

Compilador de óxido

Ahora podemos verificar si *Rust* se instaló con éxito en nuestras computadoras ejecutando el siguiente comando en nuestro terminal, si está en UNIX, o en el símbolo del sistema, si está en Windows:

```
$ rustc --version
```

Si este comando es exitoso, la versión del compilador de *Rust* instalado en nuestras computadoras se mostrará ante nuestros ojos.

Carga

Con Rust viene *Cargo* , que es una herramienta de compilación utilizada para administrar sus paquetes y proyectos de *Rust* . Para asegurarse de que esto también esté presente en su computadora, ejecute lo siguiente dentro de la consola: la consola se refiere al terminal o al símbolo del sistema según el sistema en el que esté:

```
$ cargo --version
```

Al igual que el comando equivalente para el compilador *Rust* , este regresará y mostrará la versión actual de *Cargo* .

Para crear su primer proyecto de Cargo, puede dirigirse a Cargo .

Alternativamente, puede compilar programas directamente usando `rustc` como se muestra en el ejemplo Mínimo .

Lea Empezando con Rust en línea: https://riptutorial.com/es/rust/topic/362/empezando-con-rust

Capítulo 2: Aplicaciones GUI

Introducción

Rust no tiene un marco propio para el desarrollo de GUI. Sin embargo, hay muchos enlaces a los marcos existentes. El enlace de biblioteca más avanzado es rust-gtk . Una 'semi' lista completa de enlaces se puede encontrar aquí

Examples

Simple Gtk + Ventana con texto

Agregue la dependencia de Gtk a su `Cargo.toml` :

```
[dependencies]
gtk = { git = "https://github.com/gtk-rs/gtk.git" }
```

Crea una ventana simple con lo siguiente:

```
extern crate gtk;

use gtk::prelude::*; // Import all the basic things
use gtk::{Window, WindowType, Label};

fn main() {
    if gtk::init().is_err() { //Initialize Gtk before doing anything with it
        panic!("Can't init GTK");
    }

    let window = Window::new(WindowType::Toplevel);

    //Destroy window on exit
    window.connect_delete_event(|_,_| {gtk::main_quit(); Inhibit(false) });

    window.set_title("Stackoverflow. example");
    window.set_default_size(350, 70);
    let label = Label::new(Some("Some text"));
    window.add(&label);
    window.show_all();
    gtk::main();
}
```

Ventana Gtk + con entrada y etiqueta en GtkBox, conexión de señal GtkEntry

```
extern crate gtk;

use gtk::prelude::*;
use gtk::{Window, WindowType, Label, Entry, Box as GtkBox, Orientation};

fn main() {
    if gtk::init().is_err() {
```

```
        println!("Failed to initialize GTK.");
        return;
    }

    let window = Window::new(WindowType::Toplevel);

    window.connect_delete_event(|_,_| {gtk::main_quit(); Inhibit(false) });

    window.set_title("Stackoverflow. example");
    window.set_default_size(350, 70);
    let label = Label::new(Some("Some text"));

    // Create a VBox with 10px spacing
    let bx = GtkBox::new(Orientation::Vertical, 10);
    let entry = Entry::new();

    // Connect "activate" signal to anonymous function
    // that takes GtkEntry as an argument and prints it's text
    entry.connect_activate(|x| println!("{}",x.get_text().unwrap()));

    // Add our label and entry to the box
    // Do not expand or fill, zero padding
    bx.pack_start(&label, false, false, 0);
    bx.pack_start(&entry, false, false, 0);
    window.add(&bx);
    window.show_all();
    gtk::main();
}
```

Lea Aplicaciones GUI en línea: https://riptutorial.com/es/rust/topic/7169/aplicaciones-gui

Capítulo 3: Archivo I / O

Examples

Leer un archivo en su conjunto como una cadena

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

fn main() {
    let filename = "src/main.rs";
    // Open the file in read-only mode.
    match File::open(filename) {
        // The file is open (no error).
        Ok(mut file) => {
            let mut content = String::new();

            // Read all the file content into a variable (ignoring the result of the
operation).
            file.read_to_string(&mut content).unwrap();

            println!("{}", content);

            // The file is automatically closed when is goes out of scope.
        },
        // Error handling.
        Err(error) => {
            println!("Error opening file {}: {}", filename, error);
        },
    }
}
```

Leer un archivo línea por línea

```
use std::fs::File;
use std::io::{BufRead, BufReader};

fn main() {
    let filename = "src/main.rs";
    // Open the file in read-only mode (ignoring errors).
    let file = File::open(filename).unwrap();
    let reader = BufReader::new(file);

    // Read the file line by line using the lines() iterator from std::io::BufRead.
    for (index, line) in reader.lines().enumerate() {
        let line = line.unwrap(); // Ignore errors.
        // Show the line and its number.
        println!("{}. {}", index + 1, line);
    }
}
```

Escribir en un archivo

```
use std::env;
use std::fs::File;
use std::io::Write;

fn main() {
    // Create a temporary file.
    let temp_directory = env::temp_dir();
    let temp_file = temp_directory.join("file");

    // Open a file in write-only (ignoring errors).
    // This creates the file if it does not exist (and empty the file if it exists).
    let mut file = File::create(temp_file).unwrap();

    // Write a &str in the file (ignoring the result).
    writeln!(&mut file, "Hello World!").unwrap();

    // Write a byte string.
    file.write(b"Bytes\n").unwrap();
}
```

Lee un archivo como un vec

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

fn read_a_file() -> std::io::Result<Vec<u8>> {
    let mut file = try!(File::open("example.data"));

    let mut data = Vec::new();
    try!(file.read_to_end(&mut data));

    return Ok(data);
}
```

`std::io::Result<T>` es un alias para `Result<T, std::io::Error>` .

La macro `try!()` Regresa de la función en caso de error.

`read_to_end()` es un método de `std::io::Read` rasgo, que se debe `use` explícitamente d.

`read_to_end()` no devuelve los datos que leyó. En su lugar, pone los datos en el contenedor que se le da.

Lea Archivo I / O en línea: https://riptutorial.com/es/rust/topic/1307/archivo-i---o

Capítulo 4: Argumentos de línea de comando

Introducción

La biblioteca estándar de Rust no contiene un analizador de argumentos adecuado (a diferencia de `argparse` en Python), en lugar de eso prefiere dejarlo en `argparse` de terceros. Estos ejemplos mostrarán el uso tanto de la biblioteca estándar (para formar un manejador de argumentos en bruto) como de la biblioteca de `clap` , que puede analizar los argumentos de la línea de comandos con mayor eficacia.

Sintaxis

- use std :: env; // Importar el módulo env
- let args = env :: args (); // Almacena un iterador Args en la variable args.

Examples

Usando std :: env :: args ()

Puede acceder a los argumentos de línea de comando pasados a su programa usando la función `std::env::args()` . Esto devuelve un iterador `Args` que puede hacer un bucle o recopilarlo en un `Vec` .

Iterando a través de argumentos

```
use std::env;

fn main() {
    for argument in env::args() {
        if argument == "--help" {
            println!("You passed --help as one of the arguments!");
        }
    }
}
```

Recogiendo en un `Vec`

```
use std::env;

fn main() {
    let arguments: Vec<String> = env::args().collect();
    println!("{} arguments passed", arguments.len());
}
```

Podría obtener más argumentos de los que espera si llama a su programa de esta manera:

```
./example
```

Aunque parece que no se pasaron argumentos, el primer argumento es (**generalmente**) el nombre del ejecutable. Sin embargo, esto no es una garantía, por lo que siempre debe validar y filtrar los argumentos que obtenga.

Utilizando clap

Para programas de línea de comando más grandes, usar `std::env::args()` es bastante tedioso y difícil de administrar. Puede usar `clap` para manejar su interfaz de línea de comandos, que analizará los argumentos, generará pantallas de ayuda y evitará errores.

Hay varios *patrones* que puedes usar con `clap` , y cada uno proporciona una cantidad diferente de flexibilidad.

Patrón de constructor

Este es el método más detallado (y flexible), por lo que es útil cuando necesita un control preciso de su CLI.

`clap` distingue entre *subcomandos* y *argumentos* . Los subcomandos actúan como subprogramas independientes en su programa principal, al igual `cargo run` y el `git push` . Pueden tener sus propias opciones de línea de comandos y entradas. Los argumentos son indicadores simples como `--verbose` , y pueden tomar entradas (por ejemplo, `--message "Hello, world"`)

```
extern crate clap;
use clap::{Arg, App, SubCommand};

fn main() {
    let app = App::new("Foo Server")
        .about("Serves foos to the world!")
        .version("v0.1.0")
        .author("Foo (@Example on GitHub)")
        .subcommand(SubCommand::with_name("run")
            .about("Runs the Foo Server")
            .arg(Arg::with_name("debug")
                .short("D")
                .about("Sends debug foos instead of normal foos.")))

    // This parses the command-line arguments for use.
    let matches = app.get_matches();

    // We can get the subcommand used with matches.subcommand(), which
    // returns a tuple of (&str, Option<ArgMatches>) where the &str
    // is the name of the subcommand, and the ArgMatches is an
    // ArgMatches struct:
    // https://docs.rs/clap/2.13.0/clap/struct.ArgMatches.html

    if let ("run", Some(run_matches)) = app.subcommand() {
        println!("Run was used!");
    }
}
```

Lea Argumentos de línea de comando en línea:
https://riptutorial.com/es/rust/topic/7015/argumentos-de-linea-de-comando

Capítulo 5: Arreglos, vectores y cortes

Examples

Arrays

Una matriz es una lista de objetos de un solo tipo asignados a la pila y de tamaño estático.

Los arreglos generalmente se crean al incluir una lista de elementos de un tipo dado entre corchetes. El tipo de una matriz se denota con la sintaxis especial: `[T; N]` donde `T` es el tipo de sus elementos y `N` su conteo, los cuales deben ser conocidos en el momento de la compilación.

Por ejemplo, `[4u64, 5, 6]` es una matriz de 3 elementos de tipo `[u64; 3]` . Nota: Se infiere que `5` y `6` son del tipo `u64` .

Ejemplo

```
fn main() {
    // Arrays have a fixed size.
    // All elements are of the same type.
    let array = [1, 2, 3, 4, 5];

    // Create an array of 20 elements where all elements are the same.
    // The size should be a compile-time constant.
    let ones = [1; 20];

    // Get the length of an array.
    println!("Length of ones: {}", ones.len());

    // Access an element of an array.
    // Indexing starts at 0.
    println!("Second element of array: {}", array[1]);

    // Run-time bounds-check.
    // This panics with 'index out of bounds: the len is 5 but the index is 5'.
    println!("Non existant element of array: {}", array[5]);
}
```

Limitaciones

La coincidencia de patrones en los arreglos (o segmentos) no se admite en el Rust estable (consulte # 23121 y patrones de corte).

Rust no admite la genéricoidad de los números de nivel de tipo (ver RFCs # 1657). Por lo tanto, no es posible implementar simplemente un rasgo para todas las matrices (de todos los tamaños). Como resultado, los rasgos estándar solo se implementan para arreglos de hasta un número

limitado de elementos (verificados por última vez, hasta 32 incluidos). Las matrices con más elementos son compatibles, pero no implementan los rasgos estándar (ver documentos).

Estas restricciones serán levantadas en el futuro.

Vectores

Un vector es esencialmente un puntero a una lista de objetos de un solo tipo asignados dinámicamente y de tamaño dinámico.

Ejemplo

```
fn main() {
    // Create a mutable empty vector
    let mut vector = Vec::new();

    vector.push(20);
    vector.insert(0, 10); // insert at the beginning

    println!("Second element of vector: {}", vector[1]); // 20

    // Create a vector using the `vec!` macro
    let till_five = vec![1, 2, 3, 4, 5];

    // Create a vector of 20 elements where all elements are the same.
    let ones = vec![1; 20];

    // Get the length of a vector.
    println!("Length of ones: {}", ones.len());

    // Run-time bounds-check.
    // This panics with 'index out of bounds: the len is 5 but the index is 5'.
    println!("Non existant element of array: {}", till_five[5]);
}
```

Rebanadas

Las divisiones son vistas en una lista de objetos y tienen el tipo `[T]` , que indica una porción de objetos con el tipo `T`

Una porción es un tipo sin tamaño y, por lo tanto, solo se puede usar detrás de un puntero. *(La analogía de String World: `str` , llamada segmento de cadena, también no tiene tamaño.)*

Las matrices se convierten en cortes y los vectores se pueden eliminar de referencia en los cortes. Por lo tanto, los métodos de división se pueden aplicar a ambos. *(String world analogy: `str` es `String` , lo que `[T]` es `Vec<T>` .)*

```
fn main() {
    let vector = vec![1, 2, 3, 4, 5, 6, 7, 8];
    let slice = &vector[3..6];
    println!("length of slice: {}", slice.len()); // 3
```

```
    println!("slice: {:?}", slice); // [4, 5, 6]
}
```

Lea Arreglos, vectores y cortes en línea: https://riptutorial.com/es/rust/topic/5004/arreglos--vectores-y-cortes

Capítulo 6: Asamblea en línea

Sintaxis

- #! [feature (asm)] // ¡Habilita el asm! puerta macro característica
- asm! (<template>: <output>: <input>: <clobbers>: <options>) // Emitir la plantilla de ensamblaje proporcionada (por ejemplo, "NOP", "ADD% eax, 4") con las opciones dadas.

Examples

El asm! macro

El ensamblaje en línea solo se admitirá en versiones nocturnas de Rust hasta que se estabilice . Para habilitar el uso del `asm!` macro, use el siguiente atributo de función en la parte superior del archivo principal (una *puerta de función*):

```
#![feature(asm)]
```

Entonces usa el `asm!` macro en cualquier bloque `unsafe` :

```
fn do_nothing() {
    unsafe {
        asm!("NOP");
    }

    // asm!("NOP");
    // That would be invalid here, because we are no longer in an
    // unsafe block.
}
```

Condicionar compilación en línea de montaje

Use la compilación condicional para asegurarse de que el código solo compile para el conjunto de instrucciones deseado (como `x86`). De lo contrario, el código podría no ser válido si el programa se compila para otra arquitectura, como los procesadores ARM.

```
#![feature(asm)]

// Any valid x86 code is valid for x86_64 as well. Be careful
// not to write x86_64 only code while including x86 in the
// compilation targets!
#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn do_nothing() {
    unsafe {
        asm!("NOP");
    }
}

#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
```

```
fn do_nothing() {
    // This is an alternative implementation that doesn't use any asm!
    // calls. Therefore, it should be safe to use as a fallback.
}
```

Entradas y salidas

```
#![feature(asm)]

#[cfg(any(target_arch="x86", target_arch="x86_64"))]
fn subtract(first: i32, second: i32) {
   unsafe {
        // Output values must either be unassigned (let result;) or mutable.
        let result: i32;
        // Each value that you pass in will be in a certain register, which
        // can be accessed with $0, $1, $2...
        //
        // The registers are assigned from left to right, so $0 is the
        // register containing 'result', $1 is the register containing
        // 'first' and $2 is the register containing 'second'.
        //
        // Rust uses AT&T syntax by default, so the format is:
        // SUB source, destination
        // which is equivalent to:
        // destination -= source;
        //
        // Because we want to subtract the first from the second,
        // we use the 0 constraint on 'first' to use the same
        // register as the output.
        // Therefore, we're doing:
        // SUB second, first
        // and getting the value of 'first'

        asm!("SUB $2, $0" : "=r"(result) : "0"(first), "r"(second));
        println!("{}", result);
    }
}
```

Los códigos de restricción de LLVM se pueden encontrar aquí , pero esto puede variar dependiendo de la versión de LLVM utilizada por su compilador `rustc` .

Lea Asamblea en línea en línea: https://riptutorial.com/es/rust/topic/6998/asamblea-en-linea

Capítulo 7: Auto-desreferenciación

Examples

El operador de puntos

El . Operador en Rust viene con mucha magia! Cuando usas . , el compilador insertará tantos `*` s (operaciones de desreferenciación) necesarios para encontrar el método en el "árbol" deref. Como esto sucede en el momento de la compilación, no hay un costo de tiempo de ejecución para encontrar el método.

```
let mut name: String = "hello world".to_string();
// no deref happens here because push is defined in String itself
name.push('!');

let name_ref: &String = &name;
// Auto deref happens here to get to the String. See below
let name_len = name_ref.len();
// You can think of this as syntactic sugar for the following line:
let name_len2 = (*name_ref).len();

// Because of how the deref rules work,
// you can have an arbitrary number of references.
// The . operator is clever enough to know what to do.
let name_len3 = (&&&&&&&&&&&&&name).len();
assert_eq!(name_len3, name_len);
```

La desreferenciación automática también funciona para cualquier tipo que implemente el rasgo `std::ops::Deref` .

```
let vec = vec![1, 2, 3];
let iterator = vec.iter();
```

Aquí, `iter` no es un método de `Vec<T>` , sino un método de `[T]` . Funciona porque `Vec<T>` implementa `Deref` con `Target=[T]` que le permite a `Vec<T>` convertirse en `[T]` cuando el operador `*` elimina (que el compilador puede insertar durante a .).

Deref coerciones

Dados dos tipos `T` y `U` , `&T` forzará (se convertirá implícitamente) a `&U` si y solo si `T` implementa `Deref<Target=U>`

Esto nos permite hacer cosas como esta:

```
fn foo(a: &[i32]) {
    // code
}

fn bar(s: &str) {
    // code
```

```
}

let v = vec![1, 2, 3];
foo(&v); // &Vec<i32> coerces into &[i32] because Vec<T> impls Deref<Target=[T]>

let s = "Hello world".to_string();
let rc = Rc::new(s);
// This works because Rc<T> impls Deref<Target=T> ∴ &Rc<String> coerces into
// &String which coerces into &str. This happens as much as needed at compile time.
bar(&rc);
```

Usando Deref y AsRef para argumentos de función

Para las funciones que necesitan tomar una colección de objetos, los cortes generalmente son una buena opción:

```
fn work_on_bytes(slice: &[u8]) {}
```

Porque `Vec<T>` y matrices `[T; N]` implementa `Deref<Target=[T]>` , se pueden coaccionar fácilmente a una porción:

```
let vec = Vec::new();
work_on_bytes(&vec);

let arr = [0; 10];
work_on_bytes(&arr);

let slice = &[1,2,3];
work_on_bytes(slice); // Note lack of &, since it doesn't need coercing
```

Sin embargo, en lugar de requerir explícitamente una división, se puede hacer que la función acepte cualquier tipo que *pueda* usarse como una división:

```
fn work_on_bytes<T: AsRef<[u8]>>(input: T) {
    let slice = input.as_ref();
}
```

En este ejemplo, la función `work_on_bytes` tomará cualquier tipo `T` que implemente `as_ref()` , lo que devuelve una referencia a `[u8]` .

```
work_on_bytes(vec);
work_on_bytes(arr);
work_on_bytes(slice);
work_on_bytes("strings work too!");
```

Implementación Deref para Opción y estructura de contenedor

```
use std::ops::Deref;
use std::fmt::Debug;

#[derive(Debug)]
struct RichOption<T>(Option<T>); // wrapper struct
```

```
impl<T> Deref for RichOption<T> {
    type Target = Option<T>; // Our wrapper struct will coerce into Option
    fn deref(&self) -> &Option<T> {
        &self.0 // We just extract the inner element
    }
}

impl<T: Debug> RichOption<T> {
    fn print_inner(&self) {
        println!("{:?}", self.0)
    }
}

fn main() {
    let x = RichOption(Some(1));
    println!("{:?}",x.map(|x| x + 1)); // Now we can use Option's methods...
    fn_that_takes_option(&x); // pass it to functions that take Option...
    x.print_inner() // and use it's own methods to extend Option
}

fn fn_that_takes_option<T : std::fmt::Debug>(x: &Option<T>) {
    println!("{:?}", x)
}
```

Ejemplo simple de Deref

`Deref` tiene una regla simple: si tienes un tipo `T` e implementa `Deref<Target=F>` , entonces `&T` obliga a `&F` , el compilador repetirá esto tantas veces como sea necesario para obtener F, por ejemplo:

```
fn f(x: &str) -> &str { x }
fn main() {
    // Compiler will coerce &&&&&&&str to &str and then pass it to our function
    f(&&&&&&&"It's a string");
}
```

La coacción Deref es especialmente útil cuando se trabaja con tipos de punteros, como `Box` o `Arc` , por ejemplo:

```
fn main() {
    let val = Box::new(vec![1,2,3]);
    // Now, thanks to Deref, we still
    // can use our vector method as if there wasn't any Box
    val.iter().fold(0, |acc, &x| acc + x ); // 6
    // We pass our Box to the function that takes Vec,
    // Box<Vec> coerces to Vec
    f(&val)
}

fn f(x: &Vec<i32>) {
    println!("{:?}", x) // [1,2,3]
}
```

Lea Auto-desreferenciación en línea: https://riptutorial.com/es/rust/topic/2574/auto-desreferenciacion

Capítulo 8: Bucles

Sintaxis

- loop { *bloque* } // bucle infinito
- *condición* { *bloque* }
- mientras let *patrón* = *expr* { *bloque* }
- para el *patrón* en *expr* { *bloque* } // expr debe implementar IntoIterator
- continuar // saltar al final del cuerpo del bucle, comenzando una nueva iteración si es necesario
- romper // detener el bucle
- ' *label* : loop { *block* }
- ' *label* : while *condición* { *bloque* }
- ' *label* : while let *pattern* = *expr* { *block* }
- ' *label* : para *patrón* en *expr* { *bloque* }
- continue ' *label* // salta al final de la etiqueta *etiquetada* del cuerpo del bucle, iniciando una nueva iteración si es necesario
- romper *etiqueta* // detener la etiqueta *etiquetada en* bucle

Examples

Lo esencial

Hay 4 construcciones en bucle en Rust. Todos los ejemplos a continuación producen el mismo resultado.

Bucles infinitos

```
let mut x = 0;
loop {
    if x > 3 { break; }
    println!("{}", x);
    x += 1;
}
```

Mientras bucles

```
let mut x = 0;
while x <= 3 {
    println!("{}", x);
    x += 1;
}
```

También vea: ¿Cuál es la diferencia entre `loop` y `while true` ?

Patrones combinados de patrones

Estos a veces se conocen como `while let` bucles para la brevedad.

```
let mut x = Some(0);
while let Some(v) = x {
    println!("{}", v);
    x = if v < 3 { Some(v + 1) }
        else     { None };
}
```

Esto es equivalente a una `match` dentro de un bloque de `loop` :

```
let mut x = Some(0);
loop {
    match x {
        Some(v) => {
            println!("{}", v);
            x = if v < 3 { Some(v + 1) }
                else     { None };
        }
        _       => break,
    }
}
```

Para loops

En Rust, `for` loop solo se puede usar con un objeto "iterable" (es decir, debe implementar `IntoIterator`).

```
for x in 0..4 {
    println!("{}", x);
}
```

Esto es equivalente al siguiente fragmento de código `while let` :

```
let mut iter = (0..4).into_iter();
while let Some(v) = iter.next() {
```

```
    println!("{}", v);
}
```

Nota: `0..4` devuelve un objeto `Range` que ya implementa el rasgo `Iterator` . Por `into_iter()` tanto, `into_iter()` es necesario, pero se mantiene solo para ilustrar `for` qué sirve. Para una mirada en profundidad, ver los documentos oficiales `for` Loops y `IntoIterator` .

Ver también: iteradores.

Más sobre For Loops

Como se mencionó en Conceptos básicos, podemos usar cualquier cosa que implemente `IntoIterator` con el bucle `for` :

```
let vector = vec!["foo", "bar", "baz"]; // vectors implement IntoIterator
for val in vector {
    println!("{}", val);
}
```

Rendimiento esperado:

```
foo
bar
baz
```

Tenga en cuenta que la iteración sobre el `vector` de esta manera lo consume (después del bucle `for` , el `vector` no se puede usar de nuevo). Esto se debe a que `IntoIterator::into_iter` mueve a `self` .

`IntoIterator` también se implementa con `&Vec<T>` y `&mut Vec<T>` (obteniendo valores con los tipos `&T` y `&mut T` respectivamente) para que pueda evitar el movimiento del `vector` simplemente pasándolo por referencia:

```
let vector = vec!["foo", "bar", "baz"];
for val in &vector {
    println!("{}", val);
}
println!("{:?}", vector);
```

Tenga en cuenta que `val` es de tipo `&&str` , ya que `vector` es de tipo `Vec<&str>` .

Control de bucle

Todas las construcciones de bucle permiten el uso de las declaraciones `break` y `continue` . Afectan el bucle inmediato (más interno) que lo rodea.

Control de bucle básico

`break` termina el bucle:

```
for x in 0..5 {
    if x > 2 { break; }
    println!("{}", x);
}
```

Salida

```
0
1
2
```

`continue` termina la iteración actual temprano.

```
for x in 0..5 {
    if x < 2 { continue; }
    println!("{}", x);
}
```

Salida

```
2
3
4
```

Control de bucle avanzado

Ahora, supongamos que tenemos bucles anidados y queremos `break` hacia el bucle externo. Luego, podemos usar etiquetas de bucle para especificar a qué bucle se aplica una `break` o `continue` . En el siguiente ejemplo, `'outer` es la etiqueta dada al bucle externo.

```
'outer: for i in 0..4 {
    for j in i..i+2 {
        println!("{} {}", i, j);
        if i > 1 {
            continue 'outer;
        }
    }
    println!("--");
}
```

Salida

```
0 0
0 1
--
1 1
1 2
--
2 2
3 3
```

Para `i > 1` , el bucle interno se itera sólo una vez y `--` no se imprimió.

Nota: No confunda una etiqueta de bucle con una variable de por vida. Las variables de por vida solo aparecen junto a un `&` o como un parámetro genérico dentro de `<>` .

Lea Bucles en línea: https://riptutorial.com/es/rust/topic/955/bucles

Capítulo 9: Carga

Introducción

Cargo es el administrador de paquetes de Rust, utilizado para administrar *cajas* (el término de Rust para bibliotecas / paquetes). En su mayoría, la carga obtiene paquetes de crates.io y puede administrar árboles de dependencia complejos con requisitos de versión específicos (mediante el uso de versiones semánticas). La carga también puede ayudar a construir, ejecutar y administrar proyectos de Rust con `cargo build` `cargo run` , `cargo run` `cargo test` y `cargo test` (entre otros comandos útiles).

Sintaxis

- cargo new crate_name [--bin]
- carga inicial [--bin]
- construcción de carga [--release]
- carrera de carga [--release]
- control de carga
- prueba de carga
- banco de carga
- actualización de carga
- paquete de carga
- publicación de carga
- cargo [un] instalar binary_crate_name
- búsqueda de carga
- versión de carga
- inicio de sesión de api_key

Observaciones

- En este momento, el subcomando del `cargo bench` requiere que la versión nocturna del compilador funcione de manera efectiva.

Examples

Crear nuevo proyecto

Biblioteca

```
cargo new my-library
```

Esto crea un nuevo directorio llamado `my-library` contiene el archivo de configuración de carga y

un directorio de origen que contiene un solo archivo de origen de Rust:

```
my-library/Cargo.toml
my-library/src/lib.rs
```

Estos dos archivos ya contendrán el esqueleto básico de una biblioteca, por lo que puede hacer una `cargo test` (desde el directorio de `my-library`) de inmediato para verificar si todo funciona.

Binario

```
cargo new my-binary --bin
```

Esto crea un nuevo directorio llamado `my-binary` con una estructura similar a una biblioteca:

```
my-binary/Cargo.toml
my-binary/src/main.rs
```

Esta vez, la `cargo` habrá configurado un simple binario Hello World que podremos ejecutar de inmediato con `cargo run` .

También puede crear el nuevo proyecto en el directorio actual con el subcomando `init` :

```
cargo init --bin
```

Al igual que arriba, elimine la bandera `--bin` para crear un nuevo proyecto de biblioteca. El nombre de la carpeta actual se utiliza como nombre del cajón automáticamente.

Construir proyecto

Depurar

```
cargo build
```

Lanzamiento

La `--release` con el indicador `--release` habilita ciertas optimizaciones del compilador que no se realizan al construir una compilación de depuración. Esto hace que el código se ejecute más rápido, pero también alarga un poco más el tiempo de compilación. Para un rendimiento óptimo, este comando se debe usar una vez que la versión de lanzamiento esté lista.

```
cargo build --release
```

Uso básico

```
cargo test
```

Mostrar salida del programa

```
cargo test -- --nocapture
```

Ejecutar ejemplo específico

```
cargo test test_name
```

Hola programa mundial

Esta es una sesión de shell que muestra cómo crear un programa "Hello world" y ejecutarlo con Cargo:

```
$ cargo new hello --bin
$ cd hello
$ cargo run
   Compiling hello v0.1.0 (file:///home/rust/hello)
     Running `target/debug/hello`
Hello, world!
```

Después de hacer esto, puede editar el programa abriendo `src/main.rs` en un editor de texto.

Publicando un cajón

Para publicar una caja en crates.io , debe iniciar sesión con Cargo (consulte ' *Conexión de Cargo a una cuenta de Crates.io* ').

Puedes empaquetar y publicar tu caja con los siguientes comandos:

```
cargo package
cargo publish
```

Cualquier error en su archivo `Cargo.toml` se resaltará durante este proceso. Debe asegurarse de **actualizar su versión** y de que su archivo `.gitignore` o `Cargo.toml` excluye cualquier archivo no deseado.

Conexión de carga a una cuenta Crates.io

Las cuentas en crates.io se crean al iniciar sesión con GitHub; No puedes registrarte con ningún otro método.

Para conectar su cuenta de GitHub a crates.io, haga clic en el botón ' *Iniciar sesión con GitHub* ' en la barra de menú superior y autorice a crates.io para acceder a su cuenta. Esto lo conectará a crates.io, asumiendo que todo salió bien.

Luego debe encontrar su **clave API** , que puede encontrar haciendo clic en su avatar, yendo a ' *Configuración de la cuenta* ' y copiando la línea que se ve así:

```
cargo login abcdefghijklmnopqrstuvwxyz1234567890rust
```

Debe pegarse en su terminal / línea de comando y debe autenticarlo con la instalación de `cargo` local.

Tenga cuidado con su clave API: **debe** mantenerse en secreto, como una contraseña, de lo contrario, sus jaulas podrían ser secuestradas.

Lea Carga en línea: https://riptutorial.com/es/rust/topic/1084/carga

Capítulo 10: Cierres y expresiones lambda.

Examples

Expresiones lambda simples

```
// A simple adder function defined as a lambda expression.
// Unlike with regular functions, parameter types often may be omitted because the
// compiler can infer their types
let adder = |a, b| a + b;
// Lambdas can span across multiple lines, like normal functions.
let multiplier = |a: i32, b: i32| {
    let c = b;
    let b = a;
    let a = c;
    a * b
};

// Since lambdas are anonymous functions, they can be called like other functions
println!("{}", adder(3, 5));
println!("{}", multiplier(3, 5));
```

Esto muestra:

```
8
15
```

Cierres simples

A diferencia de las funciones regulares, las expresiones lambda pueden capturar sus entornos. Tales lambdas se llaman cierres.

```

// variable definition outside the lambda expression...
let lucky_number: usize = 663;

// but the our function can access it anyway, thanks to the closures
let print_lucky_number = || println!("{}", lucky_number);

// finally call the closure
print_lucky_number();
```

Esto imprimirá:

```
663
```

Lambdas con tipos de retorno explícitos.

```
// lambda expressions can have explicitly annotated return types
```

```
let floor_func = |x: f64| -> i64 { x.floor() as i64 };
```

Pasando lambdas alrededor

Como las funciones lambda son valores en sí mismas, las almacena en colecciones, las pasa a funciones, etc. como lo haría con otros valores.

```
// This function takes two integers and a function that performs some operation on the two
arguments
fn apply_function<T>(a: i32, b: i32, func: T) -> i32 where T: Fn(i32, i32) -> i32 {
    // apply the passed function to arguments a and b
    func(a, b)
}

// let's define three lambdas, each operating on the same parameters
let sum = |a, b| a + b;
let product = |a, b| a * b;
let diff = |a, b| a - b;

// And now let's pass them to apply_function along with some arbitary values
println!("3 + 6 = {}", apply_function(3, 6, sum));
println!("-4 * 9 = {}", apply_function(-4, 9, product));
println!("7 - (-3) = {}", apply_function(7, -3, diff));
```

Esto imprimirá:

```
3 + 6 = 9
-4 * 9 = -36
7 - (-3) = 10
```

Devolviendo lambdas de funciones

Devolver lambdas (o cierres) de funciones puede ser complicado porque implementan rasgos y, por lo tanto, rara vez se conoce su tamaño exacto.

```
// Box in the return type moves the function from the stack to the heap
fn curried_adder(a: i32) -> Box<Fn(i32) -> i32> {
    // 'move' applies move semantics to a, so it can outlive this function call
    Box::new(move |b| a + b)
}

println!("3 + 4 = {}", curried_adder(3)(4));
```

Esto muestra: `3 + 4 = 7`

Lea Cierres y expresiones lambda. en línea: https://riptutorial.com/es/rust/topic/1815/cierres-y-expresiones-lambda-

Capítulo 11: Constantes asociadas

Sintaxis

- #! [feature (associated_consts)]
- const ID: i32;

Observaciones

Esta característica está disponible actualmente solo en compilador nocturno. Cuestión de seguimiento # 29646

Examples

Uso de constantes asociadas

```
// Must enable the feature to use associated constants
#![feature(associated_consts)]

use std::mem;

// Associated constants can be used to add constant attributes to types
trait Foo {
    const ID: i32;
}

// All implementations of Foo must define associated constants
// unless a default value is supplied in the definition.
impl Foo for i32 {
    const ID: i32 = 1;
}

struct Bar;

// Associated constants don't have to be bound to a trait to be defined
impl Bar {
    const BAZ: u32 = 5;
}

fn main() {
    assert_eq!(1, i32::ID);

    // The defined constant value is only stored once, so the size of
    // instances of the defined types doesn't include the constants.
    assert_eq!(4, mem::size_of::<i32>());
    assert_eq!(0, mem::size_of::<Bar>());
}
```

Lea Constantes asociadas en línea: https://riptutorial.com/es/rust/topic/7042/constantes-asociadas

Capítulo 12: Derivado personalizado: "Macros 1.1"

Introducción

Rust 1.15 agregó (estabilizó) una nueva característica: Custom derive aka Macros 1.1.

Ahora, aparte de `PartialEq` o `Debug` , puede tener `#[deriving(MyOwnDerive)]` . Dos usuarios principales de la característica son serde y diesel .

Enlace de Rust Book: https://doc.rust-lang.org/stable/book/procedural-macros.html

Examples

Verbose dumpy helloworld

Cargo.toml:

```
[package]
name = "customderive"
version = "0.1.1"

[lib]
proc-macro=true

[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

src / lib.rs:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

extern crate syn;
#[macro_use]
extern crate quote;

#[proc_macro_derive(Hello)]
pub fn qqq(input: TokenStream) -> TokenStream {
    let source = input.to_string();
    println!("Normalized source code: {}", source);
    let ast = syn::parse_derive_input(&source).unwrap();
    println!("Syn's AST: {:?}", ast); // {:#?} - pretty print
    let struct_name = &ast.ident;
    let quoted_code = quote!{
        fn hello() {
            println!("Hello, {}!", stringify!(#struct_name));
        }
```

```
    };
    println!("Quoted code: {:?}", quoted_code);
    quoted_code.parse().unwrap()
}
```

ejemplos / hello.rs:

```
#[macro_use]
extern crate customderive;

#[derive(Hello)]
struct Qqq;

fn main(){
    hello();
}
```

salida:

```
$ cargo run --example hello
   Compiling customderive v0.1.1 (file:///tmp/cd)
Normalized source code: struct Qqq;
Syn's AST: DeriveInput { ident: Ident("Qqq"), vis: Inherited, attrs: [], generics: Generics {
lifetimes: [], ty_params: [], where_clause: WhereClause { predicates: [] } }, body:
Struct(Unit) }
Quoted code: Tokens("fn hello ( ) { println ! ( \"Hello, {}!\" , stringify ! ( Qqq ) ) ; }")
warning: struct is never used: <snip>
    Finished dev [unoptimized + debuginfo] target(s) in 3.79 secs
     Running `target/x86_64-unknown-linux-gnu/debug/examples/hello`
Hello, Qqq!
```

Minimal dummy custom derive

Cargo.toml:

```
[package]
name = "customderive"
version = "0.1.0"
[lib]
proc-macro=true
```

src / lib.rs:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro_derive(Dummy)]
pub fn qqq(input: TokenStream) -> TokenStream {
    "".parse().unwrap()
}
```

ejemplos / hello.rs

```
#[macro_use]
extern crate customderive;

#[derive(Dummy)]
struct Qqq;

fn main(){}
```

Getters y setters

Cargo.toml:

```
[package]
name = "gettersetter"
version = "0.1.0"
[lib]
proc-macro=true
[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

src / lib.rs:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

extern crate syn;
#[macro_use]
extern crate quote;

#[proc_macro_derive(GetSet)]
pub fn qqq(input: TokenStream) -> TokenStream {
    let source = input.to_string();
    let ast = syn::parse_derive_input(&source).unwrap();

    let struct_name = &ast.ident;
    if let syn::Body::Struct(s) = ast.body {
        let field_names : Vec<_> = s.fields().iter().map(|ref x|
                x.ident.clone().unwrap()).collect();

        let field_getter_names = field_names.iter().map(|ref x|
                syn::Ident::new(format!("get_{}", x).as_str()));
        let field_setter_names = field_names.iter().map(|ref x|
                syn::Ident::new(format!("set_{}", x).as_str()));
        let field_types : Vec<_> = s.fields().iter().map(|ref x|
                x.ty.clone()).collect();
        let field_names2 = field_names.clone();
        let field_names3 = field_names.clone();
        let field_types2 = field_types.clone();

        let quoted_code = quote!{
            #[allow(dead_code)]
            impl #struct_name {
                #(
                    fn #field_getter_names(&self) -> &#field_types {
                        &self.#field_names2
                    }
```

```
                    fn #field_setter_names(&mut self, x : #field_types2) {
                        self.#field_names3 = x;
                    }
                )*
            }
        };
        return quoted_code.parse().unwrap();
    }
    // not a struct
    "".parse().unwrap()
}
```

ejemplos / hello.rs:

```
#[macro_use]
extern crate gettersetter;

#[derive(GetSet)]
struct Qqq {
    x : i32,
    y : String,
}

fn main(){
    let mut a = Qqq { x: 3, y: "zzaaqq".to_string() };
    println!("{}", a.get_x());
    a.set_y("123213".to_string());
    println!("{}", a.get_y());
}
```

Véase también: https://github.com/emk/accessors

Lea Derivado personalizado: "Macros 1.1" en línea: https://riptutorial.com/es/rust/topic/9104/derivado-personalizado---macros-1-1-

Capítulo 13: Documentación

Introducción

El compilador de Rust tiene varias funciones útiles para hacer que documentar su proyecto sea rápido y fácil. Puede usar lints del compilador para imponer la documentación para cada función y tener pruebas integradas en sus ejemplos.

Sintaxis

- /// Comentario de documentación externa (se aplica al artículo a continuación)
- //! Comentario interno de la documentación (se aplica al elemento adjunto)
- documento de carga # Genera documentación para esta caja de la biblioteca.
- cargo doc --open # Genera documentación para esta caja de biblioteca y navegador abierto.
- cargo doc -p CRATE # Genera documentación solo para la caja especificada.
- cargo doc --no-deps # Genera documentación para esta biblioteca y no dependencias.
- Prueba de carga # Ejecuta pruebas unitarias y pruebas de documentación.

Observaciones

Esta sección del 'Libro de óxido' puede contener información útil sobre documentación y pruebas de documentación.

Los comentarios de documentación se pueden aplicar a:

- Módulos
- Estructuras
- Enums
- Métodos
- Funciones
- Rasgos y métodos de rasgo

Examples

Documentación Lints

Para asegurarse de que todos los elementos posibles estén documentados, puede usar el enlace `missing_docs` para recibir advertencias / errores del compilador. Para recibir advertencias en toda la biblioteca, coloque este atributo en su archivo `lib.rs` :

```
#![warn(missing_docs)]
```

También puede recibir errores por falta de documentación con esta pelusa:

```
#![deny(missing_docs)]
```

De forma predeterminada, `missing_docs` , pero puede permitirlos explícitamente con este atributo:

```
#![allow(missing_docs)]
```

Puede ser útil colocarlo en un módulo para permitir la documentación faltante de un módulo, pero negarlo en todos los demás archivos.

Comentarios de documentación

Rust proporciona dos tipos de comentarios de documentación: comentarios de documentación interna y comentarios de documentación externa. A continuación se proporcionan ejemplos de cada uno.

Comentarios internos de la documentación

```
mod foo {
    //! Inner documentation comments go *inside* an item (e.g. a module or a
    //! struct). They use the comment syntax //! and must go at the top of the
    //! enclosing item.
    struct Bar {
        pub baz: i64
        //! This is invalid. Inner comments must go at the top of the struct,
        //! and must not be placed after fields.
    }
}
```

Comentarios de documentación externa

```
/// Outer documentation comments go *outside* the item that they refer to.
/// They use the syntax /// to distinguish them from inner comments.
pub enum Test {
    Success,
    Fail(Error)
}
```

Convenciones

```
/// In documentation comments, you may use **Markdown**.
/// This includes `backticks` for code, *italics* and **bold**.
/// You can add headers in your documentation, like this:
/// # Notes
/// `Foo` is unsuitable for snafucating. Use `Bar` instead.
struct Foo {
    ...
}
```

```
/// It is considered good practice to have examples in your documentation
/// under an "Examples" header, like this:
/// # Examples
/// Code can be added in "fences" of 3 backticks.
///
/// ```
/// let bar = Bar::new();
/// ```
///
/// Examples also function as tests to ensure the examples actually compile.
/// The compiler will automatically generate a main() function and run the
/// example code as a test when cargo test is run.
struct Bar {
    ...
}
```

Pruebas de documentacion

El código en la documentación de los comentarios se ejecutará automáticamente por `cargo test` . Estos se conocen como "pruebas de documentación" y ayudan a garantizar que sus ejemplos funcionen y no engañen a los usuarios de su caja.

Puede importar un pariente desde la raíz de la caja (como si hubiera un `extern crate mycrate;` oculto `extern crate mycrate;` en la parte superior del ejemplo)

```
/// ```
/// use mycrate::foo::Bar;
/// ```
```

Si su código no se ejecuta correctamente en una prueba de documentación, puede usar el atributo `no_run` , así:

```
/// ```no_run
/// use mycrate::NetworkClient;
/// NetworkClient::login("foo", "bar");
/// ```
```

También puede indicar que su código *debe* entrar en pánico, como este:

```
/// ```should_panic
/// unreachable!();
/// ```
```

Lea Documentación en línea: https://riptutorial.com/es/rust/topic/4865/documentacion

Capítulo 14: estafa

Introducción

`rustup` administra su instalación de óxido y le permite instalar diferentes versiones, que pueden configurarse e intercambiarse fácilmente.

Examples

Configurando

Instale la cadena de herramientas con

```
curl https://sh.rustup.rs -sSf | sh
```

Ya deberías tener la última versión estable de óxido. Puedes comprobarlo escribiendo

```
rustc --version
```

Si quieres actualizar, simplemente ejecuta

```
rustup update
```

Lea estafa en línea: https://riptutorial.com/es/rust/topic/8942/estafa

Capítulo 15: Estructuras

Sintaxis

- struct Foo {field1: Type1, field2: Type2}
- vamos a foo = Foo {field1: Type1 :: new (), field2: Type2 :: new ()};
- Barra de estructura (Tipo 1, Tipo 2); // tipo de tupla
- let _ = Bar (Type1 :: new (), Type2 :: new ());
- struct baz; // tipo de unidad
- dejar _ = Baz;
- vamos a Foo {field1, ..} = foo; // extraer campo1 por coincidencia de patrones
- vamos a Foo {field1: x, ..} = foo; // extraer campo1 como x
- vamos foo2 = Foo {field1: Type1 :: new (), .. foo}; // construir desde existente
- implique Foo {fn fiddle (& self) {}} // declare el método de instancia para Foo
- implique Foo {fn tweak (& mut self) {}} // declare el método de instancia mutable para Foo
- implique Foo {fn double (self) {}} // declare que posee el método de instancia para Foo
- implique Foo {fn new () {}} // declare el método asociado para Foo

Examples

Definiendo estructuras

Las estructuras en Rust se definen utilizando la palabra clave `struct` . La forma más común de estructura consiste en un conjunto de campos nombrados:

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
```

Lo anterior declara una `struct` con tres campos: `my_bool` , `my_num` y `my_string` , de los tipos `bool` , `isize` y `String` respectivamente.

Otra forma de crear `struct` en Rust es crear una *estructura de tupla* :

```
struct Bar (bool, isize, String);
```

Esto define un nuevo tipo, `Bar` , que tiene tres campos sin nombre, de tipo `bool` , `isize` y `String` , en ese orden. Esto se conoce como el patrón *newtype* , porque introduce efectivamente un nuevo "nombre" para un tipo particular. Sin embargo, lo hace de una manera más poderosa que los alias creados usando la palabra clave de `type` ; `Bar` es aquí un tipo totalmente funcional, lo que significa que puede escribir sus propios métodos para él (a continuación).

Finalmente, declare una `struct` *sin* campos, llamada *estructura tipo unidad* :

```
struct Baz;
```

Esto puede ser útil para burlarse o probar (cuando se quiere implementar un rasgo trivialmente), o como un tipo de marcador. En general, sin embargo, es poco probable que se encuentre con muchas estructuras similares a unidades.

Tenga en cuenta que los campos de `struct` en Rust son todos privados por defecto, es decir, no se puede acceder a ellos desde el código que se encuentra fuera del módulo que define el tipo. Puede prefijar un campo con la palabra clave `pub` para que el campo sea de acceso público. Además, el *propio* tipo de `struct` es privado. Para que el tipo esté disponible para otros módulos, la definición de la `struct` también debe tener el prefijo `pub` :

```
pub struct X {
    my_field: bool,
    pub our_field: bool,
}
```

Creando y utilizando valores de estructura.

Considere las siguientes definiciones de `struct` :

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
struct Bar (bool, isize, String);
struct Baz;
```

La construcción de nuevos valores de estructura para estos tipos es sencilla:

```
let foo = Foo { my_bool: true, my_num: 42, my_string: String::from("hello") };
let bar = Bar(true, 42, String::from("hello"));
let baz = Baz;
```

Acceder a los campos de una estructura usando `.` :

```
assert_eq!(foo.my_bool, true);
assert_eq!(bar.0, true); // tuple structs act like tuples
```

Un enlace mutable a una estructura puede tener sus campos mutados:

```
let mut foo = foo;
foo.my_bool = false;
let mut bar = bar;
bar.0 = false;
```

Las capacidades de coincidencia de patrones de Rust también se pueden usar para mirar dentro de una `struct` :

```
// creates bindings mb, mn, ms with values of corresponding fields in foo
let Foo { my_bool: mb, my_num: mn, my_string: ms } = foo;
assert_eq!(mn, 42);
// .. allows you to skip fields you do not care about
let Foo { my_num: mn, .. } = foo;
assert_eq!(mn, 42);
// leave out `: variable` to bind a variable by its field name
let Foo { my_num, .. } = foo;
assert_eq!(my_num, 42);
```

O haga una estructura usando una segunda estructura como "plantilla" con la *sintaxis de actualización* de Rust:

```
let foo2 = Foo { my_string: String::from("world"), .. foo };
assert_eq!(foo2.my_num, 42);
```

Métodos de estructura

Para declarar métodos en una estructura (es decir, funciones que pueden llamarse "en" la `struct` , o valores de ese tipo de `struct`), cree un bloque `impl` :

```
impl Foo {
    fn fiddle(&self) {
        // "self" refers to the value this method is being called on
        println!("fiddling {}", self.my_string);
    }
}

// ...
foo.fiddle(); // prints "fiddling hello"
```

`&self` aquí indica que es necesaria una referencia inmutable a una instancia de `struct Foo` para invocar el método de `fiddle` . Si quisiéramos modificar la instancia (como cambiar uno de sus campos), en su lugar, `&mut self` un `&mut self` (es decir, una referencia mutable):

```
impl Foo {
    fn tweak(&mut self, n: isize) {
        self.my_num = n;
    }
}

// ...
foo.tweak(43);
assert_eq!(foo.my_num, 43);
```

Finalmente, también podríamos usar `self` (tenga en cuenta la falta de un `&`) como receptor. Esto requiere que la instancia sea propiedad del llamante, y hará que la instancia se mueva al llamar al método. Esto puede ser útil si deseamos consumir, destruir o transformar completamente una instancia existente. Un ejemplo de tal caso de uso es proporcionar métodos de "encadenamiento":

```
impl Foo {
```

```
    fn double(mut self) -> Self {
        self.my_num *= 2;
        self
    }
}

// ...
foo.my_num = 1;
assert_eq!(foo.double().double().my_num, 4);
```

Tenga en cuenta que también prefijamos `self` con `mut` para que podamos mutar el self antes de devolverlo de nuevo. El tipo de retorno del método `double` también garantiza alguna explicación. `Self` dentro de un bloque `impl` refiere al tipo al que se aplica el `impl` (en este caso, `Foo`). En este caso, es principalmente una abreviatura útil para evitar volver a escribir la firma del tipo, pero en los rasgos, se puede usar para referirse al tipo subyacente que implementa un rasgo en particular.

Para declarar un *método asociado* (comúnmente denominado "método de clase" en otros idiomas) para una `struct` simplemente `struct` el argumento `self` . Dichos métodos se llaman en el propio tipo de `struct` , no en una instancia de él:

```
impl Foo {
    fn new(b: bool, n: isize, s: String) -> Foo {
        Foo { my_bool: b, my_num: n, my_string: s }
    }
}

// ...
// :: is used to access associated members of the type
let x = Foo::new(false, 0, String::from("nil"));
assert_eq!(x.my_num, 0);
```

Tenga en cuenta que los métodos de estructura solo se pueden definir para los tipos que se declararon en el módulo actual. Además, al igual que con los campos, todos los métodos de estructura son privados por defecto y, por lo tanto, solo pueden llamarse por código en el mismo módulo. Puede prefijar definiciones con la palabra clave `pub` para que sean invocables desde otro lugar.

Estructuras genéricas

Las estructuras se pueden hacer genéricas sobre uno o más parámetros de tipo. Estos tipos se incluyen entre `<>` cuando se hace referencia al tipo:

```
struct Gen<T> {
    x: T,
    z: isize,
}

// ...
let _: Gen<bool> = Gen{x: true, z: 1};
let _: Gen<isize> = Gen{x: 42, z: 2};
let _: Gen<String> = Gen{x: String::from("hello"), z: 3};
```

Se pueden dar múltiples tipos usando comas:

```
struct Gen2<T, U> {
    x: T,
    y: U,
}

// ...
let _: Gen2<bool, isize> = Gen2{x: true, y: 42};
```

Los parámetros de tipo son una parte del tipo, por lo que dos variables del mismo tipo de base, pero con parámetros diferentes, no son intercambiables:

```
let mut a: Gen<bool> = Gen{x: true, z: 1};
let b: Gen<isize> = Gen{x: 42, z: 2};
a = b; // this will not work, types are not the same
a.x = 42; // this will not work, the type of .x in a is bool
```

Si desea escribir una función que acepte una `struct` independientemente de la asignación de parámetros de tipo, esa función también debería ser genérica:

```
fn hello<T>(g: Gen<T>) {
    println!("{}", g.z); // valid, since g.z is always an isize
}
```

Pero, ¿y si quisiéramos escribir una función que siempre podría imprimir `gx` ? Tendríamos que restringir `T` para que sea de un tipo que se pueda mostrar. Podemos hacer esto con límites de tipo:

```
use std::fmt;
fn hello<T: fmt::Display>(g: Gen<T>) {
    println!("{} {}", g.x, g.z);
}
```

La función de `hello` ahora *solo se* define para las instancias `Gen` cuyo tipo `T` implementa `fmt::Display` . Si intentamos pasar un `Gen<(bool, isize)>` por ejemplo, el compilador se quejaría de que `hello` no está definido para ese tipo.

También podemos usar límites de tipo directamente en los parámetros de tipo de la `struct` para indicar que solo puedes construir esa `struct` para ciertos tipos:

```
use std::hash::Hash;
struct GenB<T: Hash> {
    x: T,
}
```

Cualquier función que tenga acceso a un `GenB` ahora sabe que el tipo de `x` implementa `Hash` y, por lo tanto, que pueden llamar a `.x.hash()` . Se pueden dar múltiples tipos de límites para el mismo parámetro separándolos con un `+` .

Igual que para las funciones, los límites de tipo se pueden colocar después de `<>` usando la

palabra clave `where` :

```
struct GenB<T> where T: Hash {
    x: T,
}
```

Esto tiene el mismo significado semántico, pero puede hacer que la firma sea más fácil de leer y formatear cuando tiene límites complejos.

Los parámetros de tipo también están disponibles para los métodos de instancia y los métodos asociados de la `struct` :

```
// note the <T> parameter for the impl as well
// this is necessary to say that all the following methods only
// exist within the context of those type parameter assignments
impl<T> Gen<T> {
    fn inner(self) -> T {
        self.x
    }
    fn new(x: T) -> Gen<T> {
        Gen{x: x}
    }
}
```

Si tiene límites de tipo en la `T Gen` , estos también deberían reflejarse en los límites de tipo de la `impl` . También puede hacer que los límites `impl` más estrictos para decir que un método determinado solo existe si el tipo satisface una propiedad en particular:

```
impl<T: Hash + fmt::Display> Gen<T> {
    fn show(&self) {
        println!("{}", self.x);
    }
}

// ...
Gen{x: 42}.show(); // works fine
let a = Gen{x: (42, true)}; // ok, because (isize, bool): Hash
a.show(); // error: (isize, bool) does not implement fmt::Display
```

Lea Estructuras en línea: https://riptutorial.com/es/rust/topic/4583/estructuras

Capítulo 16: Futuros y Async IO

Introducción

futures-rs es una biblioteca que implementa futuros y flujos de costo cero en Rust.

Los conceptos centrales de la caja de futuros son Future and Stream .

Examples

Creando un futuro con función oneshot

Hay algunas implementaciones generales de rasgos `Future` en la caja de futuros . Uno de ellos se implementa en `futures::sync::oneshot` module y está disponible a través de `futures::oneshot` function:

```
extern crate futures;

use std::thread;
use futures::Future;

fn expensive_computation() -> u32 {
    // ...
    200
}

fn main() {
    // The oneshot function returns a tuple of a Sender and a Receiver.
    let (tx, rx) = futures::oneshot();

    thread::spawn(move || {
        // The complete method resolves a values.
        tx.complete(expensive_computation());
    });

    // The map method applies a function to a value, when it is resolved.
    let rx = rx.map(|x| {
        println!("{}", x);
    });

    // The wait method blocks current thread until the value is resolved.
    rx.wait().unwrap();
}
```

Lea Futuros y Async IO en línea: https://riptutorial.com/es/rust/topic/8595/futuros-y-async-io

Capítulo 17: Generación de números aleatorios

Introducción

El óxido tiene una capacidad incorporada para proporcionar la generación de números aleatorios a través de la caja del *rand* . Una vez que formó parte de la biblioteca estándar de Rust, la funcionalidad de la caja de *rand* se separó para permitir que su desarrollo se estabilice por separado del resto del proyecto de Rust. Este tema cubrirá cómo simplemente agregar la caja de rand, luego generar y generar un número aleatorio en Rust.

Observaciones

Existe un soporte integrado para un RNG asociado con cada subproceso almacenado en el almacenamiento local de subprocesos. Se puede acceder a este RNG a través de `thread_rng` , o se puede usar implícitamente a través de `random` . Este RNG normalmente se siembra aleatoriamente desde una fuente de aleatoriedad del sistema operativo, por ejemplo, `/dev/urandom` en sistemas Unix, y se reiniciará automáticamente desde esta fuente después de generar 32 KiB de datos aleatorios.

Una aplicación que requiere una fuente de entropía para propósitos criptográficos debe usar `OsRng` , que lee la aleatoriedad de la fuente que proporciona el sistema operativo (por ejemplo, `/dev/urandom` en Unixes o `CryptGenRandom()` en Windows). Los otros generadores de números aleatorios proporcionados por este módulo no son adecuados para tales propósitos.

Examples

Generando dos números aleatorios con rand

En primer lugar, deberá agregar la caja en su archivo Cargo.toml como una dependencia.

```
[dependencies]
rand = "0.3"
```

Esto recuperará la caja de `rand` de crates.io . A continuación, agregue esto a su raíz de caja.

```
extern crate rand;
```

Como este ejemplo proporcionará una salida simple a través del terminal, crearemos una función principal e imprimiremos dos números generados aleatoriamente en la consola. El generador de números aleatorios locales de hilo se almacenará en caché en este ejemplo. Cuando se generan valores múltiples, esto a menudo puede resultar más eficiente.

```
use rand::Rng;

fn main() {

    let mut rng = rand::thread_rng();

    if rng.gen() { // random bool
        println!("i32: {}, u32: {}", rng.gen::<i32>(), rng.gen::<u32>())
    }
}
```

Cuando ejecute este ejemplo, debería ver la siguiente respuesta en la consola.

```
$ cargo run
     Running `target/debug/so`
i32: 1568599182, u32: 2222135793
```

Generando personajes con rand

Para generar caracteres, puede utilizar la función de generador de números aleatorios de subprocesos locales, `random` .

```
fn main() {
    let tuple = rand::random::<(f64, char)>();
    println!("{:?}", tuple)
}
```

Para solicitudes ocasionales o singulares, como la anterior, este es un método razonable y eficiente. Sin embargo, si pretende generar más de un puñado de números, encontrará que el generador de caché será más eficiente.

Debería esperar ver el siguiente resultado en este caso.

```
$ cargo run
     Running `target/debug/so`
(0.906881, '\u{9edc}')
```

Lea Generación de números aleatorios en línea:
https://riptutorial.com/es/rust/topic/8864/generacion-de-numeros-aleatorios

Capítulo 18: Genéricos

Examples

Declaración

```
// Generic types are declared using the <T> annotation

struct GenericType<T> {
    pub item: T
}

enum QualityChecked<T> {
    Excellent(T),
    Good(T),
    // enum fields can be generics too
    Mediocre { product: T }
}
```

Instanciación

```
// explicit type declaration
let some_value: Option<u32> = Some(13);

// implicit type declaration
let some_other_value = Some(66);
```

Parámetros de tipo múltiple

Los tipos genéricos pueden tener más de un tipo de parámetros, por ejemplo. `Result` se define así:

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

Tipos genéricos acotados

```
// Only accept T and U generic types that also implement Debug
fn print_objects<T: Debug, U: Debug>(a: T, b: U) {
    println!("A: {:?} B: {:?}", a, b);
}

print_objects(13, 44);
// or annotated explicitly
print_objects::<usize, u16>(13, 44);
```

Los límites deben cubrir todos los usos del tipo. La adición se realiza mediante el rasgo

`std::ops::Add` , que tiene parámetros de entrada y salida. `where T: std::ops::Add<u32,Output=U>` indica que es posible `Add T` a `u32` , y esta adición debe producir el tipo `U`

```
fn try_add_one<T, U>(input_value: T) -> Result<U, String>
    where T: std::ops::Add<u32,Output=U>
{
    return Ok(input_value + 1);
}
```

`Sized` límite de `Sized` está implícito por defecto. `?Sized` límite de `?Sized` permite tipos sin tamaño también.

Funciones genéricas

Las funciones genéricas permiten parametrizar algunos o todos sus argumentos.

```
fn convert_values<T, U>(input_value: T) -> Result<U, String> {
  // Try and convert the value.
  // Actual code will require bounds on the types T, U to be able to do something with them.
}
```

Si el compilador no puede inferir el parámetro de tipo, entonces puede suministrarse manualmente al llamar:

```
let result: Result<u32, String> = convert_value::<f64, u32>(13.5);
```

Lea Genéricos en línea: https://riptutorial.com/es/rust/topic/1801/genericos

Capítulo 19: Globales

Sintaxis

- IDENTIFICADOR const: tipo = constexpr;
- IDENTIFICADOR estático [mut]: tipo = expr;
- lazy_static! {IDENTIFICADOR de ref. estático: tipo = expr; }

Observaciones

- `const` valores `const` están siempre en línea y no tienen dirección en la memoria.
- `static` valores `static` nunca están en línea y tienen una instancia con una dirección fija.
- `static mut` valores `static mut` no son seguros para la memoria y, por lo tanto, solo se puede acceder a ellos en un bloque `unsafe` .
- A veces, el uso de variables mutables estáticas globales en código de subprocesos múltiples puede ser peligroso, así que considere usar std :: sync :: Mutex u otras alternativas
- `lazy_static` objetos `lazy_static` son inmutables, se inicializan solo una vez, se comparten entre todos los subprocesos y se puede acceder directamente (no hay tipos de envoltorios involucrados). En contraste, los objetos `thread_local` están diseñados para ser mutables, se inicializan una vez para cada hilo y los accesos son indirectos (que involucran el tipo de envoltorio `LocalKey<T>`)

Examples

Const

La palabra clave `const` declara un enlace constante global.

```
const DEADBEEF: u64 = 0xDEADBEEF;

fn main() {
    println("{:X}", DEADBEEF);
}
```

Esto produce

```
DEADBEEF
```

Estático

La palabra clave `static` declara un enlace estático global, que puede ser mutable.

```
static HELLO_WORLD: &'static str = "Hello, world!";

fn main() {
```

```
    println("{}", HELLO_WORLD);
}
```

Esto produce

```
Hello, world!
```

lazy_static!

Utilice la caja `lazy_static` para crear variables globales inmutables que se inicializan en tiempo de ejecución. Usamos `HashMap` como una demostración.

En `Cargo.toml` :

```
[dependencies]
lazy_static = "0.1.*"
```

En `main.rs` :

```
#[macro_use]
extern crate lazy_static;

lazy_static! {
    static ref HASHMAP: HashMap<u32, &'static str> = {
        let mut m = HashMap::new();
        m.insert(0, "hello");
        m.insert(1, ",");
        m.insert(2, " ");
        m.insert(3, "world");
        m
    };
    static ref COUNT: usize = HASHMAP.len();
}

fn main() {
    // We dereference COUNT because it's type is &usize
    println!("The map has {} entries.", *COUNT);

    // Here we don't dereference with * because of Deref coercions
    println!("The entry for `0` is \"{}\".", HASHMAP.get(&0).unwrap());
}
```

Objetos de hilo local

Un objeto de subproceso local se inicializa en su primer uso en un subproceso. Y como su nombre lo indica, cada subproceso obtendrá una copia nueva independiente de otros subprocesos.

```
use std::cell::RefCell;
use std::thread;

thread_local! {
    static FOO: RefCell<f32> = RefCell::new(1.0);
```

```
}

// When this macro expands, `FOO` gets type `thread::LocalKey<RefCell<f32>>`.
//
// Side note: One of its private member is a pointer to a function which is
// responsible for returning the thread-local object. Having all its members
// `Sync` [0], `LocalKey` is also implicitly `Sync`.
//
// [0]: As of writing this, `LocalKey` just has 2 function-pointers as members

fn main() {
    FOO.with(|foo| {
        // `foo` is of type `&RefCell<f64>`
        *foo.borrow_mut() = 3.0;
    });

    thread::spawn(move|| {
        // Note that static objects do not move (`FOO` is the same everywhere),
        // but the `foo` you get inside the closure will of course be different.
        FOO.with(|foo| {
            println!("inner: {}", *foo.borrow());
        });
    }).join().unwrap();

    FOO.with(|foo| {
        println!("main: {}", *foo.borrow());
    });
}
```

Salidas:

```
inner: 1
main: 3
```

Mut estático seguro con mut_static

Los elementos globales mutables (llamados `static mut` , que resaltan la contradicción inherente involucrada en su uso) no son seguros porque es difícil para el compilador asegurarse de que se utilicen adecuadamente.

Sin embargo, la introducción de bloqueos mutuamente exclusivos en torno a los datos permite globales mutables de memoria segura. ¡Esto NO significa que sean lógicamente seguros, sin embargo!

```
#[macro_use]
extern crate lazy_static;
extern crate mut_static;

use mut_static::MutStatic;

pub struct MyStruct { value: usize }

impl MyStruct {
    pub fn new(v: usize) -> Self{
        MyStruct { value: v }
    }
```

```
    pub fn getvalue(&self) -> usize { self.value }
    pub fn setvalue(&mut self, v: usize) { self.value = v }
}

lazy_static! {
    static ref MY_GLOBAL_STATE: MutStatic<MyStruct> = MutStatic::new();
}

fn main() {
    // Here, I call .set on the MutStatic to put data inside it.
    // This can fail.
    MY_GLOBAL_STATE.set(MyStruct::new(0)).unwrap();
    {
        // Using the global state immutably is easy...
        println!("Before mut: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }
    {
         // Using it mutably is too...
         let mut mut_handle = MY_GLOBAL_STATE.write().unwrap();
         mut_handle.setvalue(3);
         println!("Changed value to 3.");
    }
    {
        // As long as there's a scope change we can get the
        // immutable version again...
        println!("After mut: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }
    {
        // But beware! Anything can change global state!
        foo();
        println!("After foo: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }

}

// Note that foo takes no parameters
fn foo() {
    let val;
    {
        val = MY_GLOBAL_STATE.read().unwrap().getvalue();
    }
    {
        let mut mut_handle =
            MY_GLOBAL_STATE.write().unwrap();
        mut_handle.setvalue(val + 1);
    }
}
```

Este código produce la salida:

```
Before mut: 0
Changed value to 3.
After mut: 3
After foo: 4
```

Esto no es algo que debería suceder en Rust normalmente. `foo()` no tomó una referencia mutable

a nada, por lo que no debería haber mutado nada, y sin embargo lo hizo. Esto puede llevar a errores de lógica muy difíciles de depurar.

Por otro lado, esto es a veces exactamente lo que quieres. Por ejemplo, muchos motores de juegos requieren un caché global de imágenes y otros recursos que se cargan perezosamente (o utilizan alguna otra estrategia de carga compleja). MutStatic es perfecto para ese propósito.

Lea Globales en línea: https://riptutorial.com/es/rust/topic/1244/globales

Capítulo 20: Guía de estilo de óxido

Introducción

Aunque no hay una guía de estilo oficial de Rust, los siguientes ejemplos muestran las convenciones adoptadas por la mayoría de los proyectos de Rust. Seguir estas convenciones alineará el estilo de su proyecto con el de la biblioteca estándar, facilitando que las personas vean la lógica en su código.

Observaciones

Las pautas de estilo Rust oficiales estaban disponibles en el repositorio `rust-lang/rust` en GitHub, pero se han eliminado recientemente, en espera de la migración al repositorio `rust-lang-nursery/fmt-rfcs` . Hasta que se publiquen nuevas pautas allí, debe intentar seguir las pautas en el repositorio de `rust-lang` .

Puede usar rustfmt y clippy para revisar automáticamente su código en busca de problemas de estilo y formatearlo correctamente. Estas herramientas se pueden instalar usando Cargo, así:

```
cargo install clippy
cargo install rustfmt
```

Para ejecutarlos, usas:

```
cargo clippy
cargo fmt
```

Examples

Espacio en blanco

Longitud de la línea

```
// Lines should never exceed 100 characters.
// Instead, just wrap on to the next line.
let bad_example = "So this sort of code should never really happen, because it's really hard
to fit on the screen!";
```

Sangría

```
// You should always use 4 spaces for indentation.
// Tabs are discouraged - if you can, set your editor to convert
// a tab into 4 spaces.
let x = vec![1, 3, 5, 6, 7, 9];
for item in x {
    if x / 2 == 3 {
```

```
        println!("{}", x);
    }
}
```

Espacio en blanco que se arrastra

Se deben eliminar los espacios en blanco al final de los archivos o líneas.

Operadores Binarios

```
// For clarity, always add a space when using binary operators, e.g.
// +, -, =, *
let bad=3+4;
let good = 3 + 4;
```

Esto también se aplica en los atributos, por ejemplo:

```
// Good:
#[deprecated = "Don't use my class - use Bar instead!"]

// Bad:
#[deprecated="This is broken"]
```

Punto y coma

```
// There is no space between the end of a statement
// and a semicolon.

let bad = Some("don't do this!") ;
let good: Option<&str> = None;
```

Alineación de los Campos Struct

```
// Struct fields should **not** be aligned using spaces, like this:
pub struct Wrong {
    pub x  : i32,
    pub foo: i64
}

// Instead, just leave 1 space after the colon and write the type, like this:
pub struct Right {
    pub x: i32,
    pub foo: i64
}
```

Firmas de funciones

```
// Long function signatures should be wrapped and aligned so that
// the starting parameter of each line is aligned
fn foo(example_item: Bar, another_long_example: Baz,
       yet_another_parameter: Quux)
       -> ReallyLongReturnItem {
    // Be careful to indent the inside block correctly!
}
```

Tirantes

```
// The starting brace should always be on the same line as its parent.
// The ending brace should be on its own line.
fn bad()
{
    println!("This is incorrect.");
}

struct Good {
    example: i32
}

struct AlsoBad {
    example: i32 }
```

Creación de cajas

Preludios y reexportaciones

```
// To reduce the amount of imports that users need, you should
// re-export important structs and traits.
pub use foo::Client;
pub use bar::Server;
```

A veces, las cajas usan un módulo de `prelude` para contener estructuras importantes, como `std::io::prelude` . Por lo general, estos se importan con `use std::io::prelude::*;`

Importaciones

Debes ordenar tus importaciones y declaraciones así:

- declaraciones de `extern crate`
- `use` importaciones
    - Las importaciones externas de otras cajas deben ser lo primero
- Reexportaciones (`pub use`)

Nombrar

Estructuras

```
// Structs use UpperCamelCase.
pub struct Snafucator {

}

mod snafucators {
    // Try to avoid 'stuttering' by repeating
    // the module name in the struct name.

    // Bad:
    pub struct OrderedSnafucator {
```

```
    }

    // Good:
    pub struct Ordered {

    }
}
```

Rasgos

```
// Traits use the same naming principles as
// structs (UpperCamelCase).
trait Read {
    fn read_to_snafucator(&self) -> Result<(), Error>;
}
```

Cajas y módulos

```
// Modules and crates should both use snake_case.
// Crates should try to use single words if possible.
extern crate foo;
mod bar_baz {
    mod quux {

    }
}
```

Variables estáticas y constantes

```
// Statics and constants use SCREAMING_SNAKE_CASE.
const NAME: &'static str = "SCREAMING_SNAKE_CASE";
```

Enums

```
// Enum types and their variants **both** use UpperCamelCase.
pub enum Option<T> {
   Some(T),
   None
}
```

Funciones y Métodos

```
// Functions and methods use snake_case
fn snake_cased_function() {

}
```

Fijaciones variables

```
// Regular variables also use snake_case
let foo_bar = "snafu";
```

Vidas

```
// Lifetimes should consist of a single lower case letter. By
// convention, you should start at 'a, then 'b, etc.

// Good:
struct Foobar<'a> {
    x: &'a str
}

// Bad:
struct Bazquux<'stringlife> {
    my_str: &'stringlife str
}
```

Acrónimos

Los nombres de variables que contienen acrónimos, como `TCP` deben tener el siguiente estilo:

- Para los nombres de `UpperCamelCase` , la **primera letra** debe estar en mayúsculas (por ejemplo, `TcpClient`)
- Para los nombres de `snake_case` , no debe haber mayúsculas (por ejemplo, `tcp_client`)
- Para los nombres de `SCREAMING_SNAKE_CASE` , el acrónimo debe estar completamente en mayúscula (por ejemplo, `TCP_CLIENT`)

Los tipos

Escriba anotaciones

```
// There should be one space after the colon of the type
// annotation. This rule applies in variable declarations,
// struct fields, functions and methods.

// GOOD:
let mut buffer: String = String::new();
// BAD:
let mut buffer:String = String::new();
let mut buffer : String = String::new();
```

Referencias

```
// The ampersand (&) of a reference should be 'touching'
// the type it refers to.

// GOOD:
let x: &str = "Hello, world.";
// BAD:
fn fooify(x: & str) {
    println!("{}", x);
}
```

```
// Mutable references should be formatted like so:
fn bar(buf: &mut String) {

}
```

Lea Guía de estilo de óxido en línea: https://riptutorial.com/es/rust/topic/4620/guia-de-estilo-de-oxido

Capítulo 21: Instrumentos de cuerda

Introducción

A diferencia de muchos otros idiomas, Rust tiene **dos** tipos de cadena principales: `String` (un tipo de cadena asignada al montón) y `&str` (una cadena **prestada** , que no usa memoria adicional). Saber la diferencia y cuándo usar cada uno es vital para entender cómo funciona Rust.

Examples

Manipulación básica de cuerdas.

```
fn main() {
    // Statically allocated string slice
    let hello = "Hello world";

    // This is equivalent to the previous one
    let hello_again: &'static str = "Hello world";

    // An empty String
    let mut string = String::new();

    // An empty String with a pre-allocated initial buffer
    let mut capacity = String::with_capacity(10);

    // Add a string slice to a String
    string.push_str("foo");

    // From a string slice to a String
    // Note: Prior to Rust 1.9.0 the to_owned method was faster
    // than to_string. Nowadays, they are equivalent.
    let bar = "foo".to_owned();
    let qux = "foo".to_string();

    // The String::from method is another way to convert a
    // string slice to an owned String.
    let baz = String::from("foo");

    // Coerce a String into &str with &
    let baz: &str = &bar;
}
```

Nota: Tanto los métodos `String::new` como `String::with_capacity` crearán cadenas vacías. Sin embargo, este último asigna un búfer inicial, haciéndolo inicialmente más lento, pero ayudando a reducir las asignaciones posteriores. Si se conoce el tamaño final de la Cadena, se debe preferir `String::with_capacity` .

Rebanar cuerdas

```
fn main() {
    let english = "Hello, World!";
```

```
    println!("{}", &english[0..5]); // Prints "Hello"
    println!("{}", &english[7..]);  // Prints "World!"
}
```

Tenga en cuenta que necesitamos usar el operador `&` aquí. Toma una referencia y, por lo tanto, proporciona al compilador información sobre el tamaño del tipo de sector, que necesita para imprimirlo. Sin la referencia, los dos se `println!` Las llamadas serían un error en tiempo de compilación.

Advertencia: el corte funciona por **desplazamiento de byte** , no por desplazamiento de caracteres, y se producirá un error de pánico cuando los límites no estén en un límite de caracteres:

```
fn main() {
    let icelandic = "Halló, heimur!"; // note that "ó" is two-byte long in UTF-8

    println!("{}", &icelandic[0..6]); // Prints "Halló", "ó" lies on two bytes 5 and 6
    println!("{}", &icelandic[8..]);  // Prints "heimur!", the "h" is the 8th byte, but the
7th char
    println!("{}", &icelandic[0..5]); // Panics!
}
```

Esta es también la razón por la que las cadenas no admiten la indexación simple (por ejemplo, `icelandic[5]`).

Dividir una cadena

```
let strings = "bananas,apples,pear".split(",");
```

`split` devuelve un iterador.

```
for s in strings {
  println!("{}", s)
}
```

Y se puede "recopilar" en un `Vec` con el método `Iterator::collect` .

```
let strings: Vec<&str> = "bananas,apples,pear".split(",").collect(); // ["bananas", "apples",
"pear"]
```

De prestado a propiedad

```
// all variables `s` have the type `String`
let s = "hi".to_string();  // Generic way to convert into `String`. This works
                           // for all types that implement `Display`.

let s = "hi".to_owned();   // Clearly states the intend of obtaining an owned object

let s: String = "hi".into();       // Generic conversion, type annotation required
let s: String = From::from("hi");  // in both cases!
```

```
let s = String::from("hi");  // Calling the `from` impl explicitly -- the `From`
                             // trait has to be in scope!

let s = format!("hi");       // Using the formatting functionality (this has some
                             // overhead)
```

Aparte del `format!()` , Todos los métodos anteriores son igualmente rápidos.

Romper los literales de cuerda larga.

Romper las cadenas literales regulares con el `\` carácter

```
let a = "foobar";
let b = "foo\
         bar";

// `a` and `b` are equal.
assert_eq!(a,b);
```

¡Rompe los literales de cuerdas en bruto para separar cadenas y `concat!` la `concat!` macro

```
let c = r"foo\bar";
let d = concat!(r"foo\", r"bar");

// `c` and `d` are equal.
assert_eq!(c, d);
```

Lea Instrumentos de cuerda en línea: https://riptutorial.com/es/rust/topic/998/instrumentos-de-cuerda

Capítulo 22: Interfaz de función externa (FFI)

Sintaxis

- # [link (name = "snappy")] // la biblioteca extranjera a la que se vinculará (opcional)

  extern {...} // lista de firmas de funciones en la biblioteca extranjera

Examples

Llamando a la función libc del óxido nocturno

La jaula `libc` está ' característica cerrada ' y solo se puede acceder en las versiones nocturnas de Rust hasta que se considere estable.

```
#![feature(libc)]
extern crate libc;
use libc::pid_t;

#[link(name = "c")]
extern {
    fn getpid() -> pid_t;
}

fn main() {
    let x = unsafe { getpid() };
    println!("Process PID is {}", x);
}
```

Lea Interfaz de función externa (FFI) en línea: https://riptutorial.com/es/rust/topic/6140/interfaz-de-funcion-externa--ffi-

Capítulo 23: Iteradores

Introducción

Los iteradores son una poderosa característica del lenguaje en Rust, descrita por el rasgo `Iterator` . Los iteradores le permiten realizar muchas operaciones en tipos similares a colecciones, por ejemplo, `Vec<T>` , y son fáciles de componer.

Examples

Adaptadores y Consumidores

Los métodos de iterador se pueden dividir en dos grupos distintos:

Adaptadores

Los adaptadores toman un iterador y devuelven otro iterador

```
//          Iterator  Adapter
//              |       |
let my_map = (1..6).map(|x| x * x);
println!("{:?}", my_map);
```

Salida

```
Map { iter: 1..6 }
```

Tenga en cuenta que los valores no se enumeraron, lo que indica que los iteradores no se evalúan con entusiasmo; los iteradores son "perezosos".

Los consumidores

Los consumidores toman un iterador y devuelven algo más que un iterador, consumiendo el iterador en el proceso.

```
//                    Iterator  Adapter       Consumer
//                        |       |             |
let my_squares: Vec<_> = (1..6).map(|x| x * x).collect();
println!("{:?}", my_squares);
```

Salida

```
[1, 4, 9, 16, 25]
```

Otros ejemplos de consumidores incluyen `find` , `fold` y `sum` .

```
let my_squared_sum: u32 = (1..6).map(|x| x * x).sum();
println!("{:?}", my_squared_sum);
```

Salida

```
55
```

Una breve prueba de primalidad.

```
fn is_prime(n: u64) -> bool {
    (2..n).all(|divisor| n % divisor != 0)
}
```

Por supuesto que esto no es una prueba rápida. Podemos dejar de probar en la raíz cuadrada de `n`:

```
(2..n)
    .take_while(|divisor| divisor * divisor <= n)
    .all(|divisor| n % divisor != 0)
```

Iterador personalizado

```
struct Fibonacci(u64, u64);

impl Iterator for Fibonacci {
    type Item = u64;

    // The method that generates each item
    fn next(&mut self) -> Option<Self::Item> {
        let ret = self.0;
        self.0 = self.1;
        self.1 += ret;

        Some(ret) // since `None` is never returned, we have an infinite iterator
    }

    // Implementing the `next()` method suffices since every other iterator
    // method has a default implementation
}
```

Ejemplo de uso:

```
// the iterator method `take()` is an adapter which limits the number of items
// generated by the original iterator
for i in Fibonacci(0, 1).take(10) {
    println!("{}", i);
}
```

Lea Iteradores en línea: https://riptutorial.com/es/rust/topic/4657/iteradores

Capítulo 24: La coincidencia de patrones

Sintaxis

- _ // patrón de comodín, coincide con cualquier cosa[1]
- *ident* // patrón de unión, coincide con cualquier cosa y lo une a *ident*
- *ident* @ *pat* // igual que arriba, pero permite que coincida aún más con lo que está enlazado
- ref *ident* // patrón de encuadernación, coincide con cualquier cosa y lo enlaza con un *ident* *de* referencia [1]
- ref mut *ident* // patrón de unión, coincide con cualquier cosa y lo une a una referencia de mutable *ident* [1]
- & *pat* // coincide con una referencia (*pat no* es, por lo tanto, una referencia, sino el árbitro)
- & mut *pat* // igual que el anterior con una referencia mutable[1]
- *CONST* // coincide con una constante nombrada
- *Struct* { *field1* , *field2* } // coincide y deconstruye un valor de estructura, vea más abajo la nota sobre campos[1]
- *EnumVariant* // coincide con una variante de enumeración
- *EnumVariant* (*pat1* , *pat2*) // coincide con una variante de enumeración y los parámetros correspondientes
- *EnumVariant* (*pat1* , *pat2* , .., *patn*) // igual que el anterior, pero omite todos los parámetros menos el primero, el segundo y el último
- (*pat1* , *pat2*) // coincide con una tupla y los elementos correspondientes[1]
- (*pat1* , *pat2* , .., *patn*) // igual que arriba pero salta todos los elementos menos el primero, el segundo y el último[1]
- *encendido* // coincide con una constante literal (char, tipos numéricos, booleano y cadena)
- *pat1* ... *pat2* // coincide con un valor en ese rango (inclusive) (tipos de caracteres y numéricos)

Observaciones

Al deconstruir un valor de estructura, el campo debe tener el formato `field_name` o `field_name : pattern` . Si no se especifica ningún patrón, se realiza un enlace implícito:

```
let Point { x, y } = p;
// equivalent to
let Point { x: x, y: y } = p;

let Point { ref x, ref y } = p;
// equivalent to
let Point { x: ref x, y: ref y } = p;
```

1: patrón irrefutable

Examples

Patrón de coincidencia con enlaces

Es posible vincular valores a nombres usando `@` :

```
struct Badger {
    pub age: u8
}

fn main() {
    // Let's create a Badger instances
    let badger_john = Badger { age: 8 };

    // Now try to find out what John's favourite activity is, based on his age
    match badger_john.age {
        // we can bind value ranges to variables and use them in the matched branches
        baby_age @ 0...1 => println!("John is {} years old, he sleeps a lot", baby_age),
        young_age @ 2...4 => println!("John is {} years old, he plays all day", young_age),
        adult_age @ 5...10 => println!("John is {} years old, he eats honey most of the time",
adult_age),
        old_age => println!("John is {} years old, he mostly reads newspapers", old_age),
    }
}
```

Esto imprimirá:

```
John is 8 years old, he eats honey most of the time
```

Coincidencia de patrones básicos

```
// Create a boolean value
let a = true;

// The following expression will try and find a pattern for our value starting with
// the topmost pattern.
// This is an exhaustive match expression because it checks for every possible value
match a {
  true => println!("a is true"),
  false => println!("a is false")
}
```

Si no cubrimos todos los casos, obtendremos un error de compilación:

```
match a {
  true => println!("most important case")
}
// error: non-exhaustive patterns: `false` not covered [E0004]
```

Podemos usar _ como el caso predeterminado / comodín, coincide con todo:

```
// Create an 32-bit unsigned integer
let b: u32 = 13;

match b {
```

```
  0 => println!("b is 0"),
  1 => println!("b is 1"),
  _ => println!("b is something other than 0 or 1")
}
```

Este ejemplo imprimirá:

```
a is true
b is something else than 0 or 1
```

Coincidencia de patrones múltiples

Es posible tratar múltiples valores distintos de la misma manera, usando `|` :

```
enum Colour {
    Red,
    Green,
    Blue,
    Cyan,
    Magenta,
    Yellow,
    Black
}

enum ColourModel {
    RGB,
    CMYK
}

// let's take an example colour
let colour = Colour::Red;

let model = match colour {
    // check if colour is any of the RGB colours
    Colour::Red | Colour::Green | Colour::Blue => ColourModel::RGB,
    // otherwise select CMYK
    _ => ColourModel::CMYK,
};
```

Patrón condicional que coincide con los guardias

Los patrones se pueden hacer coincidir en función de los valores independientes del valor que se hace coincidir utilizando `if` guardias:

```
// Let's imagine a simplistic web app with the following pages:
enum Page {
  Login,
  Logout,
  About,
  Admin
}

// We are authenticated
let is_authenticated = true;
```

```
// But we aren't admins
let is_admin = false;

let accessed_page = Page::Admin;

match accessed_page {
    // Login is available for not yet authenticated users
    Page::Login if !is_authenticated => println!("Please provide a username and a password"),

    // Logout is available for authenticated users
    Page::Logout if is_authenticated => println!("Good bye"),

    // About is a public page, anyone can access it
    Page::About => println!("About us"),

    // But the Admin page is restricted to administators
    Page::Admin if is_admin => println!("Welcome, dear administrator"),

    // For every other request, we display an error message
    _ => println!("Not available")
}
```

Esto mostrará *"No disponible"* .

si let / while let

---

if let

Combina una `match` patrón y una declaración `if` , y permite que se realicen breves coincidencias no exhaustivas.

```
if let Some(x) = option {
    do_something(x);
}
```

Esto es equivalente a:

```
match option {
    Some(x) => do_something(x),
    _ => {},
}
```

Estos bloques también pueden tener `else` declaraciones.

```
if let Some(x) = option {
    do_something(x);
} else {
    panic!("option was None");
}
```

Este bloque es equivalente a:

```
match option {
    Some(x) => do_something(x),
    None => panic!("option was None"),
}
```

`while let`

Combina una coincidencia de patrón y un bucle while.

```
let mut cs = "Hello, world!".chars();
while let Some(x) = cs.next() {
    print("{}+", x);
}
println!("");
```

Esto imprime `H+e+l+l+o+,+ +w+o+r+l+d+!+` .

Es equivalente a usar un `loop {}` y una declaración de `match` :

```
let mut cs = "Hello, world!".chars();
loop {
    match cs.next() {
        Some(x) => print("{}+", x),
        _ => break,
    }
}
println!("");
```

Extraer referencias de patrones.

A veces es necesario poder extraer valores de un objeto utilizando solo referencias (es decir, sin transferir la propiedad).

```
struct Token {
  pub id: u32
}

struct User {
  pub token: Option<Token>
}


fn main() {
    // Create a user with an arbitrary token
    let user = User { token: Some(Token { id: 3 }) };

    // Let's borrow user by getting a reference to it
    let user_ref = &user;

    // This match expression would not compile saying "cannot move out of borrowed
    // content" because user_ref is a borrowed value but token expects an owned value.
    match user_ref {
        &User { token } => println!("User token exists? {}", token.is_some())
    }
```

```
    // By adding 'ref' to our pattern we instruct the compiler to give us a reference
    // instead of an owned value.
    match user_ref {
        &User { ref token } => println!("User token exists? {}", token.is_some())
    }

    // We can also combine ref with destructuring
    match user_ref {
        // 'ref' will allow us to access the token inside of the Option by reference
        &User { token: Some(ref user_token) } => println!("Token value: {}", user_token.id ),
        &User { token: None } => println!("There was no token assigned to the user" )
    }

    // References can be mutable too, let's create another user to demonstrate this
    let mut other_user = User { token: Some(Token { id: 4 }) };

    // Take a mutable reference to the user
    let other_user_ref_mut = &mut other_user;

    match other_user_ref_mut {
        // 'ref mut' gets us a mutable reference allowing us to change the contained value
directly.
        &mut User { token: Some(ref mut user_token) } => {
            user_token.id = 5;
            println!("New token value: {}", user_token.id )
        },
        &mut User { token: None } => println!("There was no token assigned to the user" )
    }
}
```

Se imprimirá esto:

```
User token exists? true
Token value: 3
New token value: 5
```

Lea La coincidencia de patrones en línea: https://riptutorial.com/es/rust/topic/1188/la-coincidencia-de-patrones

Capítulo 25: Macros

Observaciones

Puede encontrar un tutorial de macros en *The Rust Programming Language* (también conocido como The Book) .

Examples

Tutorial

Las macros nos permiten abstraer patrones sintácticos que se repiten muchas veces. Por ejemplo:

```
/// Computes `a + b * c`. If any of the operation overflows, returns `None`.
fn checked_fma(a: u64, b: u64, c: u64) -> Option<u64> {
    let product = match b.checked_mul(c) {
        Some(p) => p,
        None => return None,
    };
    let sum = match a.checked_add(product) {
        Some(s) => s,
        None => return None,
    };
    Some(sum)
}
```

Notamos que las dos declaraciones de `match` son muy similares: ambas tienen el mismo patrón

```
match expression {
    Some(x) => x,
    None => return None,
}
```

Imagina que representamos el patrón anterior como `try_opt!(expression)` , luego podríamos reescribir la función en solo 3 líneas:

```
fn checked_fma(a: u64, b: u64, c: u64) -> Option<u64> {
    let product = try_opt!(b.checked_mul(c));
    let sum = try_opt!(a.checked_add(product));
    Some(sum)
}
```

`try_opt!` no se puede escribir una función porque una función no admite el retorno anticipado. Pero podríamos hacerlo con una macro: siempre que tengamos estos patrones sintácticos que no se pueden representar usando una función, podemos tratar de usar una macro.

Definimos una macro usando las `macro_rules!` sintaxis:

```
macro_rules! try_opt {
//                  ^ note: no `!` after the macro name
    ($e:expr) => {
//   ^~~~~~~ The macro accepts an "expression" argument, which we call `$e`.
//           All macro parameters must be named like `$xxxxx`, to distinguish from
//           normal tokens.
        match $e {
//            ^~ The input is used here.
            Some(x) => x,
            None => return None,
        }
    }
}
```

¡Eso es! Hemos creado nuestra primera macro.

(Pruébalo en Rust Playground)

Crear una macro HashSet

```
// This example creates a macro `set!` that functions similarly to the built-in
// macro vec!

use std::collections::HashSet;

macro_rules! set {
    ( $( $x:expr ),* ) => {  // Match zero or more comma delimited items
        {
            let mut temp_set = HashSet::new();  // Create a mutable HashSet
            $(
                temp_set.insert($x); // Insert each item matched into the HashSet
            )*
            temp_set // Return the populated HashSet
        }
    };
}

// Usage
let my_set = set![1, 2, 3, 4];
```

Recursion

Una macro puede llamarse a sí misma, como una función recursiva:

```
macro_rules! sum {
    ($base:expr) => { $base };
    ($a:expr, $($rest:expr),+) => { $a + sum!($($rest),+) };
}
```

¡Vamos por la expansión de la `sum!(1, 2, 3)` :

```
   sum!(1, 2, 3)
//      ^  ^~~~
//     $a  $rest
=> 1 + sum!(2, 3)
```

```
//          ^  ^
//         $a  $rest
=> 1 + (2 + sum!(3))
//               ^
//               $base
=> 1 + (2 + (3))
```

Límite de recursión

Cuando el compilador está expandiendo las macros demasiado profundamente, se rendirá. De forma predeterminada, el compilador fallará después de expandir las macros a 64 niveles de profundidad, por lo que, por ejemplo, la siguiente expansión causará un error:

```
sum!(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
     21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,
     41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62)

// error: recursion limit reached while expanding the macro `sum`
//  --> <anon>:3:46
// 3 |>         ($a:expr, $($rest:expr),+) => { $a + sum!($($rest),+) };
//   |>                                              ^^^^^^^^^^^^^^^^
```

Cuando se alcanza un límite de recursión, debería considerar refactorizar su macro, por ejemplo,

- ¿Tal vez la recursión podría ser reemplazada por la repetición?
- Tal vez el formato de entrada podría cambiarse a algo menos sofisticado, por lo que no necesitamos recursión para que coincida.

Si hay alguna razón legítima para que 64 niveles no sean suficientes, siempre puede aumentar el límite de la caja invocando la macro con el atributo:

```
#![recursion_limit="128"]
//                  ^~~ set the recursion limit to 128 levels deep.
```

Patrones multiples

Una macro puede producir diferentes salidas contra diferentes patrones de entrada:

```
/// The `sum` macro may be invoked in two ways:
///
///     sum!(iterator)
///     sum!(1234, iterator)
///
macro_rules! sum {
    ($iter:expr) => {    // This branch handles the `sum!(iterator)` case
        $iter.fold(0, |a, b| a + *b)
    };
//   ^ use `;` to separate each branch
    ($start:expr, $iter:expr) => {  // This branch handles the `sum!(1234, iter)` case
        $iter.fold($start, |a, b| a + *b)
    };
}
```

```
fn main() {
    assert_eq!(10, sum!([1, 2, 3, 4].iter()));
    assert_eq!(23, sum!(6, [2, 5, 9, 1].iter()));
}
```

Especificadores de fragmentos - Tipo de patrones

En `$e:expr` , el `expr` se llama el *especificador de fragmento* . Le dice al analizador qué tipo de tokens está esperando el parámetro `$e` . Rust proporciona una variedad de especificadores de fragmentos para permitir que la entrada sea muy flexible.

Especificador	Descripción	Ejemplos
`ident`	Identificador	`x` , `foo`
`path`	Nombre calificado	`std::collection::HashSet` , `Vec::new`
`ty`	Tipo	`i32` , `&T` , `Vec<(char, String)>`
`expr`	Expresión	`2+2` , `f(42)` , `if true { 1 } else { 2 }`
`pat`	Modelo	`_` , `c @ 'a' ... 'z'` , `(true, &x)` , `Badger { age, .. }`
`stmt`	Declaración	`let x = 3` , `return 42`
`block`	Bloque delimitado por llaves	`{ foo(); bar(); }` , `{ x(); y(); z() }`
`item`	ít	`fn foo() {}` , `struct Bar;` , `use std::io;`
`meta`	Dentro del atributo	`cfg!(windows)` , `doc="comment"`
`tt`	Árbol simbólico	`+` , `foo` , `5` , `[?!(???)]`

Tenga en cuenta que un comentario doc `/// comment` se trata de la misma forma que `#[doc="comment"]` a una macro.

```
macro_rules! declare_const_option_type {
    (
        $(#[$attr:meta])*
        const $name:ident: $ty:ty as optional;
    ) => {
        $(#[$attr])*
        const $name: Option<$ty> = None;
    }
}

declare_const_option_type! {
    /// some doc comment
    const OPT_INT: i32 as optional;
```

```
}

// The above will be expanded to:
#[doc="some doc comment"]
const OPT_INT: Option<i32> = None;
```

Seguir set

Algunos especificadores de fragmentos requieren que el token que sigue debe ser uno de un conjunto restringido, denominado "conjunto de seguimiento". Esto permite cierta flexibilidad para que la sintaxis de Rust evolucione sin romper las macros existentes.

Especificador	Seguir set
`expr` , `stmt`	`=>` , `;`
`ty` , `path`	`=>` , `=` | `;` `:` `>` `[` `{` `as` `where`
`pat`	`=>` , `=` | `if` `in`
`ident` , `block` , `item` , `meta` , `tt`	*cualquier token*

```
macro_rules! invalid_macro {
    ($e:expr + $f:expr) => { $e + $f };
//           ^
//           `+` is not in the follow set of `expr`,
//           and thus the compiler will not accept this macro definition.
    ($($e:expr)/+) => { $($e)/+ };
//             ^
//             The separator `/` is not in the follow set of `expr`
//             and thus the compiler will not accept this macro definition.
}
```

Exportando e importando macros

Exportando una macro para permitir que otros módulos la utilicen:

```
    #[macro_export]
//  ^~~~~~~~~~~~~~~ Think of it as `pub` for macros.
    macro_rules! my_macro { (..) => {..} }
```

Usando macros de otras cajas o módulos:

```
   #[macro_use] extern crate lazy_static;
// ^~~~~~~~~~~~ Must add this in order to use macros from other crates

   #[macro_use] mod macros;
// ^~~~~~~~~~~~ The same for child modules.
```

Depuración de macros

(Todos estos son inestables y, por lo tanto, solo pueden usarse desde un compilador nocturno).

log_syntax! ()

```
#![feature(log_syntax)]

macro_rules! logged_sum {
    ($base:expr) => {
        { log_syntax!(base = $base); $base }
    };
    ($a:expr, $($rest:expr),+) => {
        { log_syntax!(a = $a, rest = $($rest),+); $a + logged_sum!($($rest),+) }
    };
}

const V: u32 = logged_sum!(1, 2, 3);
```

Durante la compilación, imprimirá lo siguiente en la salida estándar:

a = 1, resto = 2, 3
a = 2, resto = 3
base = 3

--muy expandido

Ejecuta el compilador con:

```
rustc -Z unstable-options --pretty expanded filename.rs
```

Esto expandirá todas las macros y luego imprimirá el resultado expandido en la salida estándar, por ejemplo, lo anterior probablemente generará:

```
#![feature(prelude_import)]
#![no_std]
#![feature(log_syntax)]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std as std;


const V: u32 = { false; 1 + { false; 2 + { false; 3 } } };
```

(Esto es similar a la bandera `-E` en los compiladores de C `gcc` y `clang` .)

Lea Macros en línea: https://riptutorial.com/es/rust/topic/1031/macros

Capítulo 26: Manejo de errores

Introducción

El óxido utiliza los valores del `Result<T, E>` para indicar errores recuperables durante la ejecución. Los errores irrecuperables causan pánicos, que es un tema propio.

Observaciones

Los detalles del manejo de errores se describen en *The Rust Programming Language* (también conocido como The Book)

Examples

Métodos de resultados comunes

```
use std::io::{Read, Result as IoResult};
use std::fs::File;

struct Config(u8);

fn read_config() -> IoResult<String> {
    let mut s = String::new();
    let mut file = File::open(&get_local_config_path())
        // or_else closure is invoked if Result is Err.
        .or_else(|_| File::open(&get_global_config_path()))?;
    // Note: In `or_else`, the closure should return a Result with a matching
    //       Ok type, whereas in `and_then`, the returned Result should have a
    //       matching Err type.
    let _ = file.read_to_string(&mut s)?;
    Ok(s)
}

struct ParseError;

fn parse_config(conf_str: String) -> Result<Config, ParseError> {
    // Parse the config string...
    if conf_str.starts_with("bananas") {
        Err(ParseError)
    } else {
        Ok(Config(42))
    }
}

fn run() -> Result<(), String> {
    // Note: The error type of this function is String. We use map_err below to
    //       make the error values into String type
    let conf_str = read_config()
        .map_err(|e| format!("Failed to read config file: {}", e))?;
    // Note: Instead of using `?` above, we can use `and_then` to wrap the let
    //       expression below.
    let conf_val = parse_config(conf_str)
```

```
        .map(|Config(v)| v / 2) // map can be used to map just the Ok value
        .map_err(|_| "Failed to parse the config string!".to_string())?;

    // Run...

    Ok(())
}

fn main() {
    match run() {
        Ok(_) => println!("Bye!"),
        Err(e) => println!("Error: {}", e),
    }
}

fn get_local_config_path() -> String {
    let user_config_prefix = "/home/user/.config";
    // code to get the user config directory
    format!("{}/my_app.rc", user_config_prefix)
}

fn get_global_config_path() -> String {
    let global_config_prefix = "/etc";
    // code to get the global config directory
    format!("{}/my_app.rc", global_config_prefix)
}
```

Si los archivos de configuración no existen, esto genera:

```
Error: Failed to read config file: No such file or directory (os error 2)
```

Si falla el análisis, esto genera:

```
Error: Failed to parse the config string!
```

Nota: A medida que el proyecto crezca, será incómodo manejar los errores con estos métodos básicos (documentos) sin perder información sobre el origen y la ruta de propagación de los errores. Además, definitivamente es una mala práctica convertir errores en cadenas prematuramente para manejar múltiples tipos de errores como se muestra arriba. Una forma mucho mejor es usar la `error-chain` la caja.

Tipos de error personalizados

```
use std::error::Error;
use std::fmt;
use std::convert::From;
use std::io::Error as IoError;
use std::str::Utf8Error;

#[derive(Debug)] // Allow the use of "{:?}" format specifier
enum CustomError {
    Io(IoError),
    Utf8(Utf8Error),
    Other,
}
```

```
// Allow the use of "{}" format specifier
impl fmt::Display for CustomError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            CustomError::Io(ref cause) => write!(f, "I/O Error: {}", cause),
            CustomError::Utf8(ref cause) => write!(f, "UTF-8 Error: {}", cause),
            CustomError::Other => write!(f, "Unknown error!"),
        }
    }
}

// Allow this type to be treated like an error
impl Error for CustomError {
    fn description(&self) -> &str {
        match *self {
            CustomError::Io(ref cause) => cause.description(),
            CustomError::Utf8(ref cause) => cause.description(),
            CustomError::Other => "Unknown error!",
        }
    }

    fn cause(&self) -> Option<&Error> {
        match *self {
            CustomError::Io(ref cause) => Some(cause),
            CustomError::Utf8(ref cause) => Some(cause),
            CustomError::Other => None,
        }
    }
}

// Support converting system errors into our custom error.
// This trait is used in `try!`.
impl From<IoError> for CustomError {
    fn from(cause: IoError) -> CustomError {
        CustomError::Io(cause)
    }
}
impl From<Utf8Error> for CustomError {
    fn from(cause: Utf8Error) -> CustomError {
        CustomError::Utf8(cause)
    }
}
```

Iterando a través de las causas.

A menudo es útil para propósitos de depuración encontrar la causa raíz de un error. Para examinar un valor de error que implemente `std::error::Error` :

```
use std::error::Error;

let orig_error = call_returning_error();

// Use an Option<&Error>. This is the return type of Error.cause().
let mut err = Some(&orig_error as &Error);

// Print each error's cause until the cause is None.
while let Some(e) = err {
    println!("{}", e);
```

```
    err = e.cause();
}
```

Informes de errores básicos y manejo

`Result<T, E>` es un tipo de `enum` que tiene dos variantes: `Ok(T)` indica una ejecución exitosa con un resultado significativo de tipo `T` , y `Err(E)` indica la ocurrencia de un error inesperado durante la ejecución, descrito por un valor de tipo `E` .

```
enum DateError {
    InvalidDay,
    InvalidMonth,
}

struct Date {
    day: u8,
    month: u8,
    year: i16,
}

fn validate(date: &Date) -> Result<(), DateError> {
    if date.month < 1 || date.month > 12 {
        Err(DateError::InvalidMonth)
    } else if date.day < 1 || date.day > 31 {
        Err(DateError::InvalidDay)
    } else {
        Ok(())
    }
}

fn add_days(date: Date, days: i32) -> Result<Date, DateError> {
    validate(&date)?; // notice `?` -- returns early on error
    // the date logic ...
    Ok(date)
}
```

También ver documentos para más detalles sobre `?` operador.

La biblioteca estándar contiene un rasgo de `Error` que se recomienda implementar en todos los tipos de error. A continuación se muestra un ejemplo de implementación.

```
use std::error::Error;
use std::fmt;

#[derive(Debug)]
enum DateError {
    InvalidDay(u8),
    InvalidMonth(u8),
}

impl fmt::Display for DateError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            &DateError::InvalidDay(day) => write!(f, "Day {} is outside range!", day),
            &DateError::InvalidMonth(month) => write!(f, "Month {} is outside range!", month),
        }
```

```
    }
}

impl Error for DateError {
    fn description(&self) -> &str {
        match self {
            &DateError::InvalidDay(_) => "Day is outside range!",
            &DateError::InvalidMonth(_) => "Month is outside range!",
        }
    }

    // cause method returns None by default
}
```

Nota: En general, la `Option<T>` no debe utilizarse para informar errores. `Option<T>` indica una posibilidad esperada de no existencia de un valor y una sola razón directa para ello. En contraste, el `Result<T, E>` se usa para informar errores inesperados durante la ejecución, y especialmente cuando hay múltiples modos de falla para distinguirlos. Además, el `Result<T, E>` solo se utiliza como valores de retorno. (Una vieja discusión.)

Lea Manejo de errores en línea: https://riptutorial.com/es/rust/topic/1762/manejo-de-errores

Capítulo 27: Manejo de señales

Observaciones

El óxido no tiene una forma adecuada, idiomática y segura de vincularse con las señales del sistema operativo, pero hay algunas cajas que proporcionan el manejo de la señal, pero son altamente *experimentales* e *inseguras,* así que tenga cuidado al usarlas.

Sin embargo, hay una discusión en el repositorio rust-lang / rfcs sobre cómo implementar el manejo de señales nativas para el óxido.

Discusión sobre RFCs: https://github.com/rust-lang/rfcs/issues/1368

Examples

Manejo de señal con caja de señal de chan.

La caja de señal de chan proporciona una solución para manejar la señal del sistema operativo utilizando canales, aunque esta caja es **experimental** y debe usarse con **cuidado** .

Ejemplo tomado de BurntSushi / chan-signal .

```
#[macro_use]
extern crate chan;
extern crate chan_signal;

use chan_signal::Signal;

fn main() {
    // Signal gets a value when the OS sent a INT or TERM signal.
    let signal = chan_signal::notify(&[Signal::INT, Signal::TERM]);
    // When our work is complete, send a sentinel value on `sdone`.
    let (sdone, rdone) = chan::sync(0);
    // Run work.
    ::std::thread::spawn(move || run(sdone));

    // Wait for a signal or for work to be done.
    chan_select! {
        signal.recv() -> signal => {
            println!("received signal: {:?}", signal)
        },
        rdone.recv() => {
            println!("Program completed normally.");
        }
    }
}

fn run(_sdone: chan::Sender<()>) {
    println!("Running work for 5 seconds.");
    println!("Can you send a signal quickly enough?");
    // Do some work.
    ::std::thread::sleep_ms(5000);
```

```
    // _sdone gets dropped which closes the channel and causes `rdone`
    // to unblock.
}
```

Manipulación de señales con caja nix.

La caja nix proporciona una API de Rust de UNIX para manejar señales, sin embargo, requiere el uso de óxido **no seguro,** por lo que debe tener **cuidado** .

```
use nix::sys::signal;

extern fn handle_sigint(_:i32) {
    // Be careful here...
}

fn main() {
    let sig_action = signal::SigAction::new(handle_sigint,
                                            signal::SockFlag::empty(),
                                            signal::SigSet::empty());
    signal::sigaction(signal::SIGINT, &sig_action);
}
```

Ejemplo de Tokio

La caja de tokio-signal proporciona una solución basada en tokio para el manejo de señales. Sin embargo, todavía está en sus primeras etapas.

```
extern crate futures;
extern crate tokio_core;
extern crate tokio_signal;

use futures::{Future, Stream};
use tokio_core::reactor::Core
use tokio_signal::unix::{self as unix_signal, Signal};
use std::thread::{self, sleep};
use std::time::Duration;
use std::sync::mpsc::{channel, Receiver};

fn run(signals: Receiver<i32>) {
    loop {
        if let Some(signal) = signals.try_recv() {
            eprintln!("received {} signal");
        }
        sleep(Duration::from_millis(1));
    }
}

fn main() {
    // Create channels for sending and receiving signals
    let (signals_tx, signals_rx) = channel();

    // Execute the program with the receiving end of the channel
    // for listening to any signals that are sent to it.
    thread::spawn(move || run(signals_rx));
```

```
    // Create a stream that will select over SIGINT, SIGTERM, and SIGHUP signals.
    let signals = Signal::new(unix_signal::SIGINT, &handle).flatten_stream()
        .select(Signal::new(unix_signal::SIGTERM, &handle).flatten_stream())
        .select(Signal::new(unix_signal::SIGHUP, &handle).flatten_stream());

    // Execute the event loop that will listen for and transmit received
    // signals to the shell.
    core.run(signals.for_each(|signal| {
        let _ = signals_tx.send(signal);
        Ok(())
    })).unwrap();
}
```

Lea Manejo de señales en línea: https://riptutorial.com/es/rust/topic/3995/manejo-de-senales

Capítulo 28: Marco web de hierro

Introducción

Iron es un marco web popular para Rust (basado en la biblioteca Hyper de nivel inferior) que promueve la idea de extensibilidad a través de *middleware* . Gran parte de la funcionalidad necesaria para crear un sitio web útil se puede encontrar en el middleware de Iron en lugar de en la propia biblioteca.

Examples

Sencillo servidor 'Hello'

Este ejemplo envía una respuesta codificada al usuario cuando envía una solicitud de servidor.

```
extern crate iron;

use iron::prelude::*;
use iron::status;

// You can pass the handler as a function or a closure. In this
// case, we've chosen a function for clarity.
// Since we don't care about the request, we bind it to _.
fn handler(_: &mut Request) -> IronResult<Response> {
    Ok(Response::with((status::Ok, "Hello, Stack Overflow")))
}

fn main() {
    Iron::new(handler).http("localhost:1337").expect("Server failed!")
}
```

Al crear un nuevo servidor `Iron` en este ejemplo, `expect` detectar cualquier error con un mensaje de error más descriptivo. En las aplicaciones de producción, *maneje el error* producido (consulte la documentación de `http()`).

Instalación de hierro

Agregue esta dependencia al archivo `Cargo.toml` :

```
[dependencies]
iron = "0.4.0"
```

Run `cargo build` y Cargo descargará e instalará la versión especificada de Iron.

Enrutamiento simple con hierro

Este ejemplo proporcionará enrutamiento web básico utilizando Iron.

Para empezar, deberá agregar la dependencia de Iron a su archivo `Cargo.toml` .

```
[dependencies]
iron = "0.4.*"
```

Usaremos la propia librería Router de Iron. Para simplificar, el proyecto Iron proporciona esta biblioteca como parte de la biblioteca de Iron Core, eliminando cualquier necesidad de agregarla como una dependencia separada. A continuación, hacemos referencia tanto a la biblioteca Iron como a la biblioteca Router.

```
extern crate iron;
extern crate router;
```

Luego importamos los objetos requeridos para permitirnos administrar el enrutamiento y devolver una respuesta al usuario.

```
use iron::{Iron, Request, Response, IronResult};
use iron::status;
use router::{Router};
```

En este ejemplo, lo mantendremos simple escribiendo la lógica de enrutamiento dentro de nuestra función `main()` . Por supuesto, a medida que su aplicación crezca, querrá separar el enrutamiento, el registro, los problemas de seguridad y otras áreas de su aplicación web. Por ahora, este es un buen punto de partida.

```
fn main() {
    let mut router = Router::new();
    router.get("/", handler, "handler");
    router.get("/:query", query_handler, "query_handler");
```

Repasemos lo que hemos logrado hasta ahora. Actualmente, nuestro programa crea una instancia de un nuevo objeto de Iron `Router` y adjunta dos "controladores" a dos tipos de solicitud de URL: el primero (`"/"`) es la raíz de nuestro dominio, y el segundo (`"/:query"`) es cualquier ruta debajo de la raiz

Al usar un punto y coma antes de la palabra "consulta", le estamos diciendo a Iron que tome esta parte de la ruta de la URL como una variable y la pase a nuestro controlador.

La siguiente línea de código es cómo creamos una instancia de Iron, designando nuestro propio objeto `router` para administrar nuestras solicitudes de URL. El dominio y el puerto están codificados en este ejemplo para simplificar.

```
    Iron::new(router).http("localhost:3000").unwrap();
```

A continuación, declaramos dos funciones en línea que son nuestros manejadores, `handler` y `query_handler` . Ambos se utilizan para demostrar URL fijas y URL variables.

En la segunda función tomamos la variable `"query"` de la URL que contiene el objeto de solicitud, y la enviamos de vuelta al usuario como respuesta.

```
    fn handler(_: &mut Request) -> IronResult<Response> {
        Ok(Response::with((status::Ok, "OK")))
    }

    fn query_handler(req: &mut Request) -> IronResult<Response> {
        let ref query = req.extensions.get::<Router>()
            .unwrap().find("query").unwrap_or("/");
        Ok(Response::with((status::Ok, *query)))
    }
}
```

Si ejecutamos este ejemplo, podremos ver el resultado en el navegador web en `localhost:3000` . La raíz del dominio debe responder con `"OK"` , y cualquier cosa debajo de la raíz debe repetir la ruta de regreso.

El siguiente paso de este ejemplo podría ser la separación del enrutamiento y el servicio de páginas estáticas.

Lea Marco web de hierro en línea: https://riptutorial.com/es/rust/topic/8060/marco-web-de-hierro

Capítulo 29: Módulos

Sintaxis

- mod *modname* ; // Buscar el módulo en *modname* .rs o *modname* /mod.rs en el mismo directorio
- mod *modname* { *bloque* }

Examples

Árbol de módulos

Archivos:

```
- example.rs (root of our modules tree, generally named lib.rs or main.rs when using Cargo)
- first.rs
- second/
  - mod.rs
  - sub.rs
```

Módulos:

```
- example        -> example
  - first        -> example::first
  - second       -> example::second
    - sub        -> example::second::sub
  - third        -> example::third
```

`example.rs`

```
pub mod first;
pub mod second;
pub mod third {
    ...
}
```

El `second` módulo que se debe declarar en el archivo `example.rs` como su padre es `example` y no, por ejemplo, `first` y, por lo tanto, no se puede declarar en el `first.rs` u otro archivo en el mismo nivel de directorio

`second/mod.rs`

```
pub mod sub;
```

El atributo # [ruta]

El atributo `#[path]` Rust se puede usar para especificar la ruta para buscar un módulo en particular si no está en la ubicación estándar. Sin embargo, esto generalmente no se recomienda , ya que

hace que la jerarquía de módulos sea frágil y facilita la ruptura de la compilación al mover un archivo en un directorio completamente diferente.

```
#[path="../path/to/module.rs"]
mod module;
```

Nombres en código vs nombres en `use`

La sintaxis de dos puntos de los nombres en la declaración de `use` es similar a los nombres usados en otras partes del código, pero el significado de estas rutas es diferente.

Los nombres en la declaración de `use` por defecto se interpretan como absolutos, comenzando en la raíz de la caja. Los nombres en otras partes del código son relativos al módulo actual.

La declaración:

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

tiene el mismo significado en el archivo principal de la caja, así como en los módulos. Por otro lado, un nombre de función como `std::fs::File::open()` se referirá a la biblioteca estándar de Rust solo en el archivo principal de la caja, porque los nombres en el código se interpretan en relación con el módulo actual.

```
fn main() {
    std::fs::File::open("example"); // OK
}

mod my_module {
   fn my_fn() {
       // Error! It means my_module::std::fs::File::open()
       std::fs::File::open("example");

       // OK. `::` prefix makes it absolute
       ::std::fs::File::open("example");

       // OK. `super::` reaches out to the parent module, where `std` is present
       super::std::fs::File::open("example");
   }
}
```

Para hacer que `std::…` nombres se comporten en todas partes igual que en la raíz de la caja, usted podría agregar:

```
use std;
```

A la inversa, puede hacer que las rutas de `use` relativas prefijándolas con palabras clave `self` o `super` :

```
 use self::my_module::my_fn;
```

Accediendo al Módulo de Padres

A veces, puede ser útil importar funciones y estructuras relativamente sin tener que `use` algo con su ruta absoluta en su proyecto. Para lograr esto, puedes usar el módulo `super` , así:

```
fn x() -> u8 {
    5
}

mod example {
    use super::x;

    fn foo() {
        println!("{}", x());
    }
}
```

Puede usar `super` varias veces para llegar al "abuelo" de su módulo actual, pero debe tener cuidado de no introducir problemas de legibilidad si usa `super` demasiadas veces en una importación.

Exportaciones y Visibilidad

Estructura de directorios:

```
yourproject/
    Cargo.lock
    Cargo.toml
    src/
        main.rs
        writer.rs
```

main.rs

```
// This is import from writer.rs
mod writer;

fn main() {
    // Call of imported write() function.
    writer::write()

    // BAD
    writer::open_file()
}
```

escritor.r

```
// This function WILL be exported.
pub fn write() {}

// This will NOT be exported.
fn open_file() {}
```

Organización del código básico

Veamos cómo podemos organizar el código, cuando el código base se está haciendo más grande.

01. Funciones

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

02. Módulos - En el mismo archivo

```
fn main() {
  greet::hello();
}

mod greet {
  // By default, everything inside a module is private
  pub fn hello() { // So function has to be public to access from outside
    println!("Hello, world!");
  }
}
```

03. Módulos - En un archivo diferente en el mismo directorio

Cuando mueva algo de código a un archivo nuevo, no es necesario envolver el código en una declaración de `mod` . El archivo mismo actúa como un módulo.

```
// ↳ main.rs
mod greet; // import greet module

fn main() {
  greet::hello();
}
```

```
// ↳ greet.rs
pub fn hello() { // function has to be public to access from outside
  println!("Hello, world!");
}
```

> Cuando mueva algo de código a un nuevo archivo, si ese código ha sido ajustado desde una declaración `mod` , será un submódulo del archivo.

```
// ↳ main.rs
mod greet;

fn main() {
  greet::hello::greet();
```

```
}
```

```
// ↳ greet.rs
pub mod hello { // module has to be public to access from outside
  pub fn greet() { // function has to be public to access from outside
    println!("Hello, world!");
  }
}
```

04. Módulos - En un archivo diferente en un directorio diferente

Cuando mueva algún código a un nuevo archivo en un directorio diferente, el directorio mismo actúa como un módulo. Y `mod.rs` en la raíz del módulo es el punto de entrada al módulo de directorio. Todos los demás archivos en ese directorio, actúan como un submódulo de ese directorio.

```
// ↳ main.rs
mod greet;

fn main() {
  greet::hello();
}
```

```
// ↳ greet/mod.rs
pub fn hello() {
  println!("Hello, world!");
}
```

Cuando tienes varios archivos en la raíz del módulo,

```
// ↳ main.rs
mod greet;

fn main() {
  greet::hello_greet()
}
```

```
// ↳ greet/mod.rs
mod hello;

pub fn hello_greet() {
  hello::greet()
}
```

```
// ↳ greet/hello.rs
pub fn greet() {
  println!("Hello, world!");
}
```

05. Módulos - Con `self`

```
fn main() {
  greet::call_hello();
```

```
}

mod greet {
  pub fn call_hello() {
    self::hello();
  }

  fn hello() {
    println!("Hello, world!");
  }
}
```

06. Módulos - Con `super`

1. Cuando desee acceder a una función raíz desde dentro de un módulo,

```
fn main() {
  dash::call_hello();
}

fn hello() {
  println!("Hello, world!");
}

mod dash {
  pub fn call_hello() {
    super::hello();
  }
}
```

2. Cuando desee acceder a una función en el módulo externo / principal desde un módulo anidado,

```
fn main() {
  outer::inner::call_hello();
}

mod outer {

  pub fn hello() {
    println!("Hello, world!");
  }

  mod inner {
    pub fn call_hello() {
      super::hello();
    }
  }

}
```

07. Módulos - Con `use`

1. Cuando desee enlazar la ruta completa a un nuevo nombre,

```
use greet::hello::greet as greet_hello;
```

```
fn main() {
  greet_hello();
}

mod greet {
  pub mod hello {
    pub fn greet() {
      println!("Hello, world!");
    }
  }
}
```

2. Cuando quieras usar el contenido del nivel de alcance del cajón

```
fn main() {
  user::hello();
}

mod greet {
  pub mod hello {
    pub fn greet() {
      println!("Hello, world!");
    }
  }
}

mod user {
  use greet::hello::greet as call_hello;

  pub fn hello() {
    call_hello();
  }
}
```

Lea Módulos en línea: https://riptutorial.com/es/rust/topic/2528/modulos

Capítulo 30: Opción

Introducción

El tipo `Option<T>` es el equivalente de Rust de los tipos anulables, sin todos los problemas que lo acompañan. La mayoría de los lenguajes tipo C permiten que cualquier variable sea `null` si no hay datos presentes, pero el tipo de `Option` está inspirado en lenguajes funcionales que favorecen a los 'opcionales' (por ejemplo, la mónada de Haskell, `Maybe`). El uso de los tipos de `Option` le permitirá expresar la idea de que los datos pueden o no estar allí (ya que Rust no tiene tipos anulables).

Examples

Creando un valor de opción y una coincidencia de patrón

```
// The Option type can either contain Some value or None.
fn find(value: i32, slice: &[i32]) -> Option<usize> {
    for (index, &element) in slice.iter().enumerate() {
        if element == value {
            // Return a value (wrapped in Some).
            return Some(index);
        }
    }
    // Return no value.
    None
}

fn main() {
    let array = [1, 2, 3, 4, 5];
    // Pattern match against the Option value.
    if let Some(index) = find(2, &array) {
        // Here, there is a value.
        println!("The element 2 is at index {}.", index);
    }

    // Check if the result is None (no value).
    if let None = find(12, &array) {
        // Here, there is no value.
        println!("The element 12 is not in the array.");
    }

    // You can also use `is_some` and `is_none` helpers
    if find(12, &array).is_none() {
        println!("The element 12 is not in the array.");
    }
}
```

Destructurando una opción

```
fn main() {
    let maybe_cake = Some("Chocolate cake");
```

```
    let not_cake = None;

    // The unwrap method retrieves the value from the Option
    // and panics if the value is None
    println!("{}", maybe_cake.unwrap());

    // The expect method works much like the unwrap method,
    // but panics with a custom, user provided message.
    println!("{}", not_cake.expect("The cake is a lie."));

    // The unwrap_or method can be used to provide a default value in case
    // the value contained within the option is None. This example would
    // print "Cheesecake".
    println!("{}", not_cake.unwrap_or("Cheesecake"));

    // The unwrap_or_else method works like the unwrap_or method,
    // but allows us to provide a function which will return the
    // fallback value. This example would print "Pumpkin Cake".
    println!("{}", not_cake.unwrap_or_else(|| { "Pumpkin Cake" }));

    // A match statement can be used to safely handle the possibility of none.
    match maybe_cake {
        Some(cake) => println!("{} was consumed.", cake),
        None       => println!("There was no cake.")
    }

    // The if let statement can also be used to destructure an Option.
    if let Some(cake) = maybe_cake {
        println!("{} was consumed.", cake);
    }
}
```

Desenvolver una referencia a una opción que posee su contenido

Una referencia a una opción `&Option<T>` no se puede desempaquetar si el tipo `T` no se puede copiar. La solución es cambiar la opción a `&Option<&T>` usando `as_ref()` .

La oxidación prohíbe la transferencia de la propiedad de los objetos mientras se prestan los objetos. Cuando la propia Opción se toma prestada (`&Option<T>`), su contenido también es, indirectamente, prestado.

```
#[derive(Debug)]
struct Foo;

fn main() {
    let wrapped = Some(Foo);
    let wrapped_ref = &wrapped;

    println!("{:?}", wrapped_ref.unwrap()); // Error!
}
```

no puede salir del contenido prestado [--explain E0507]

Sin embargo, es posible crear una referencia al contenido de la `Option<T>` . El método `as_ref()` Option devuelve una opción para `&T` , que puede ser desenvuelta sin transferencia de propiedad:

```
println!("{:?}", wrapped_ref.as_ref().unwrap());
```

Usando la opción con el mapa y and_then

La operación de `map` es una herramienta útil cuando se trabaja con matrices y vectores, pero también se puede utilizar para tratar los valores de las `Option` de una manera funcional.

```
fn main() {

    // We start with an Option value (Option<i32> in this case).
    let some_number = Some(9);

    // Let's do some consecutive calculations with our number.
    // The crucial point here is that we don't have to unwrap
    // the content of our Option type - instead, we're just
    // transforming its content. The result of the whole operation
    // will still be an Option<i32>. If the initial value of
    // 'some_number' was 'None' instead of 9, then the result
    //  would also be 'None'.
    let another_number = some_number
        .map(|n| n - 1) // => Some(8)
        .map(|n| n * n) // => Some(64)
        .and_then(|n| divide(n, 4)); // => Some(16)

    // In the last line above, we're doing a division using a helper
    // function (definition: see bottom).
    // 'and_then' is very similar to 'map', but allows us to pass a
    // function which returns an Option type itself. To ensure that we
    // don't end up with Option<Option<i32>>, 'and_then' flattens the
    // result (in other languages, 'and_then' is also known as 'flatmap').

    println!("{}", to_message(another_number));
    // => "16 is definitely a number!"

    // For the sake of completeness, let's check the result when
    // dividing by zero.
    let final_number = another_number
        .and_then(|n| divide(n, 0)); // => None

    println!("{}", to_message(final_number));
    // => "None!"
}

// Just a helper function for integer division. In case
// the divisor is zero, we'll get 'None' as result.
fn divide(number: i32, divisor: i32) -> Option<i32> {
    if divisor != 0  { Some(number/divisor) } else { None }
}

// Creates a message that tells us whether our
// Option<i32> contains a number or not. There are other
// ways to achieve the same result, but let's just use
// map again!
fn to_message(number: Option<i32>) -> String {
    number
        .map(|n| format!("{} is definitely a number!", n)) // => Some("...")
        .unwrap_or("None!".to_string()) // => "..."
}
```

Lea Opción en línea: https://riptutorial.com/es/rust/topic/1125/opcion

Capítulo 31: Operadores y sobrecargas

Introducción

La mayoría de los operadores en Rust se pueden definir ("sobrecargar") para los tipos definidos por el usuario. Esto se puede lograr implementando el rasgo respectivo en el módulo `std::ops` .

Examples

Sobrecarga del operador de suma (+)

Sobrecargar el operador de suma (+) requiere implementar el rasgo `std::ops::Add` .

De la documentación, la definición completa del rasgo es:

```
pub trait Add<RHS = Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

¿Como funciona?

- El rasgo se implementa para el tipo del lado izquierdo.
- el rasgo se implementa para *un* argumento del lado derecho, a menos que se especifique de manera predeterminada que tenga el mismo tipo que el lado izquierdo
- el tipo de resultado de la adición se especifica en el tipo asociado `Output`

Por lo tanto, tener 3 tipos diferentes es posible.

Nota: el rasgo que se consume son los argumentos del lado izquierdo y del lado derecho, es posible que prefiera implementarlo para referencias a su tipo en lugar de a los tipos básicos.

Implementando `+` para un tipo personalizado:

```
use std::ops::Add;

#[derive(Clone)]
struct List<T> {
    data: Vec<T>,
}
//  Implementation which consumes both LHS and RHS
impl<T> Add for List<T> {
    type Output = List<T>;

    fn add(self, rhs: List<T>) -> List<T> {
        self.data.extend(rhs.data.drain(..));
        self
    }
}
```

```
//  Implementation which only consumes RHS (and thus where LHS != RHS)
impl<'a, T: Clone> Add<List<T>> for &'a List<T> {
    type Output = List<T>;

    fn add(self, rhs: List<T>) -> List<T> {
        self.clone() + rhs
    }
}
```

Lea Operadores y sobrecargas en línea: https://riptutorial.com/es/rust/topic/7271/operadores-y-sobrecargas

Capítulo 32: Óxido de metal desnudo

Introducción

La biblioteca estándar de Rust (`std`) se compila contra solo un puñado de arquitecturas. Por lo tanto, para compilar en otras arquitecturas (que soporta LLVM), los programas Rust podrían optar por no usar todo el `std` , y en su lugar usar solo el subconjunto portátil, conocido como The Core Library (`core`).

Examples

#! [no_std] Hola, Mundo!

```
#![feature(start, libc, lang_items)]
#![no_std]
#![no_main]

// The libc crate allows importing functions from C.
extern crate libc;

// A list of C functions that are being imported
extern {
    pub fn printf(format: *const u8, ...) -> i32;
}

#[no_mangle]
// The main function, with its input arguments ignored, and an exit status is returned
pub extern fn main(_nargs: i32, _args: *const *const u8) -> i32 {
    // Print "Hello, World" to stdout using printf
    unsafe {
        printf(b"Hello, World!\n" as *const u8);
    }

    // Exit with a return status of 0.
    0
}

#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] extern fn panic_fmt() -> ! { panic!() }
```

Lea Óxido de metal desnudo en línea: https://riptutorial.com/es/rust/topic/8344/oxido-de-metal-desnudo

Capítulo 33: Pánicos y Desenrollamientos

Introducción

Cuando los programas de Rust alcanzan un estado en el que se ha producido un error crítico, el `panic!` Se puede llamar a la macro para salir rápidamente (a menudo comparada, pero sutilmente diferente, a una excepción en otros idiomas). El manejo adecuado de errores debe involucrar tipos de `Result` , aunque esta sección solo tratará el `panic!` y sus conceptos.

Observaciones

Los pánicos no siempre causan pérdidas de memoria u otras pérdidas de recursos. De hecho, los pánicos generalmente conservan las invariantes de RAII, ejecutando los destructores (Implementaciones de Drop) de las estructuras a medida que la pila se desenrolla. Sin embargo, si hay un segundo pánico durante este proceso, el programa simplemente se cancela; en ese punto, las garantías invariantes de RAII son nulas.

Examples

Trata de no entrar en pánico

En Rust, hay dos métodos principales para indicar que algo salió mal en un programa: una función que devuelve un `Err(E)` (potencialmente definido por el `Err(E)` , del tipo `Result<T, E>` y un `panic!` .

El pánico **no** es una alternativa para las excepciones, que se encuentran comúnmente en otros idiomas. En Rust, un pánico es para indicar que algo salió mal y que no puede continuar. Aquí hay un ejemplo de la fuente de `Vec` para `push` :

```
pub fn push(&mut self, value: T) {
    // This will panic or abort if we would allocate > isize::MAX bytes
    // or if the length increment would overflow for zero-sized types.
    if self.len == self.buf.cap() {
        self.buf.double();
    }
    ...
}
```

Si nos quedamos sin memoria, no hay mucho más que Rust pueda hacer, por lo que puede entrar en pánico (el comportamiento predeterminado) o abortar (que debe configurarse con un indicador de compilación).

El pánico desenrollará la pila, ejecutará los destructores y garantizará que se limpie la memoria. Abort no hace esto, y confía en el sistema operativo para limpiarlo correctamente.

Trate de ejecutar el siguiente programa normalmente y con

```
[profile.dev]
panic = "abort"
```

en su `Cargo.toml` .

```
// main.rs
struct Foo(i32);
impl Drop for Foo {
    fn drop(&mut self) {
        println!("Dropping {:?}!", self.0);
    }
}
fn main() {
    let foo = Foo(1);
    panic!("Aaaaaaahhhhh!");
}
```

Lea Pánicos y Desenrollamientos en línea: https://riptutorial.com/es/rust/topic/6895/panicos-y-desenrollamientos

Capítulo 34: Paralelismo

Introducción

El paralelismo es bien soportado por la biblioteca estándar de Rust a través de varias clases como el módulo `std::thread` , canales y atomics. Esta sección lo guiará a través del uso de estos tipos.

Examples

Comenzando un nuevo hilo

Para iniciar un nuevo hilo:

```
use std::thread;

fn main() {
    thread::spawn(move || {
        // The main thread will not wait for this thread to finish. That
        // might mean that the next println isn't even executed before the
        // program exits.
        println!("Hello from spawned thread");
    });

    let join_handle = thread::spawn(move || {
        println!("Hello from second spawned thread");
        // To ensure that the program waits for a thread to finish, we must
        // call `join()` on its join handle. It is even possible to send a
        // value to a different thread through the join handle, like the
        // integer 17 in this case:
        17
    });

    println!("Hello from the main thread");

    // The above three printlns can be observed in any order.

    // Block until the second spawned thread has finished.
    match join_handle.join() {
        Ok(x) => println!("Second spawned thread returned {}", x),
        Err(_) => println!("Second spawned thread panicked")
    }
}
```

Comunicación entre hilos con canales

Los canales se pueden utilizar para enviar datos de un hilo a otro. A continuación se muestra un ejemplo de un sistema simple productor-consumidor, donde el hilo principal produce los valores 0, 1, ..., 9 y el hilo generado los imprime:

```
use std::thread;
```

```
use std::sync::mpsc::channel;

fn main() {
    // Create a channel with a sending end (tx) and a receiving end (rx).
    let (tx, rx) = channel();

    // Spawn a new thread, and move the receiving end into the thread.
    let join_handle = thread::spawn(move || {
        // Keep receiving in a loop, until tx is dropped!
        while let Ok(n) = rx.recv() { // Note: `recv()` always blocks
            println!("Received {}", n);
        }
    });

    // Note: using `rx` here would be a compile error, as it has been
    // moved into the spawned thread.

    // Send some values to the spawned thread. `unwrap()` crashes only if the
    // receiving end was dropped before it could be buffered.
    for i in 0..10 {
        tx.send(i).unwrap(); // Note: `send()` never blocks
    }

    // Drop `tx` so that `rx.recv()` returns an `Err(_)`.
    drop(tx);

    // Wait for the spawned thread to finish.
    join_handle.join().unwrap();
}
```

Comunicación entre hilos con tipos de sesión

Los tipos de sesión son una forma de informar al compilador sobre el protocolo que desea utilizar para comunicarse entre subprocesos, no como protocolo en HTTP o FTP, sino el patrón de flujo de información entre subprocesos. Esto es útil ya que el compilador ahora evitará que rompas accidentalmente tu protocolo y causes interbloqueos o bloqueos vitales entre los hilos, algunos de los problemas más notoriamente difíciles de depurar, y una fuente importante de Heisenbugs. Los tipos de sesión funcionan de manera similar a los canales descritos anteriormente, pero puede ser más intimidante comenzar a usar. Aquí hay una comunicación simple de dos hilos:

```
// Session Types aren't part of the standard library, but are part of this crate.
// You'll need to add session_types to your Cargo.toml file.
extern crate session_types;

// For now, it's easiest to just import everything from the library.
use session_types::*;

// First, we describe what our client thread will do. Note that there's no reason
// you have to use a client/server model - it's just convenient for this example.
// This type says that a client will first send a u32, then quit. `Eps` is
// shorthand for "end communication".
// Session Types use two generic parameters to describe the protocol - the first
// for the current communication, and the second for what will happen next.
type Client = Send<u32, Eps>;
// Now, we define what the server will do: it will receive as u32, then quit.
type Server = Recv<u32, Eps>;
```

```
// This function is ordinary code to run the client. Notice that it takes
// ownership of a channel, just like other forms of interthread communication -
// but this one about the protocol we just defined.
fn run_client(channel: Chan<(), Client>) {
    let channel = channel.send(42);
    println!("The client just sent the number 42!");
    channel.close();
}

// Now we define some code to run the server. It just accepts a value and prints
// it.
fn run_server(channel: Chan<(), Server>) {
    let (channel, data) = channel.recv();
    println!("The server received some data: {}", data);
    channel.close();
}

fn main() {
    // First, create the channels used for the two threads to talk to each other.
    let (server_channel, client_channel) = session_channel();

    // Start the server on a new thread
    let server_thread = std::thread::spawn(move || {
        run_server(server_channel);
    });

    // Run the client on this thread.
    run_client(client_channel);

    // Wait for the server to finish.
    server_thread.join().unwrap();
}
```

Debe observar que el método principal se ve muy similar al método principal para la comunicación entre hilos definidos anteriormente, si el servidor se movió a su propia función. Si tuvieras que ejecutar esto, obtendrías la salida:

```
The client just sent the number 42!
The server received some data: 42
```

en ese orden.

¿Por qué pasar por todas las molestias de definir los tipos de cliente y servidor? ¿Y por qué redefinimos el canal en el cliente y el servidor? Estas preguntas tienen la misma respuesta: ¡el compilador nos impedirá romper el protocolo! Si el cliente intentara recibir datos en lugar de enviarlos (lo que resultaría en un interbloqueo en el código ordinario), el programa no se compilaría, ya que el objeto de canal del cliente *no tiene un método de* `recv`. Además, si intentamos definir el protocolo de una manera que pudiera provocar un interbloqueo (por ejemplo, si el cliente y el servidor intentaron recibir un valor), la compilación fallaría cuando creamos los canales. Esto se debe a que `Send` y `Recv` son "Tipos duales", lo que significa que si el Servidor hace uno, el Cliente tiene que hacer el otro, si ambos intentan `Recv` , estaremos en problemas. `Eps` es su propio tipo dual, ya que está bien que tanto el Cliente como el Servidor acuerden cerrar el canal.

Por supuesto, cuando hacemos alguna operación en el canal, pasamos a un nuevo estado en el

protocolo, y las funciones disponibles podrían cambiar, por lo que tenemos que redefinir el enlace del canal. Por suerte, `session_types` se encarga de eso y siempre devuelve el nuevo canal (excepto el `close` , en cuyo caso no hay un nuevo canal). Esto también significa que todos los métodos en un canal también se apropian del canal, por lo que si olvida redefinir el canal, el compilador también le dará un error al respecto. Si suelta un canal sin cerrarlo, también es un error de tiempo de ejecución (desafortunadamente, es imposible verificarlo en el momento de la compilación).

Hay muchos más tipos de comunicación que solo `Send` y `Recv` ; por ejemplo, `Offer` le da al otro lado del canal la posibilidad de elegir entre dos posibles ramas del protocolo, y `Rec` y `Var` trabajan juntos para permitir bucles y recursión en el protocolo. . Muchos más ejemplos de tipos de sesión y otros tipos están disponibles en el `session_types` GitHub repositorio . La documentación de la biblioteca se puede encontrar aquí.

Atómica y ordenación de la memoria

Los tipos atómicos son los bloques de construcción de estructuras de datos sin bloqueo y otros tipos concurrentes. Se debe especificar un orden de memoria, que representa la resistencia de la barrera de memoria, al acceder / modificar un tipo atómico. Rust proporciona 5 primitivas de ordenación de memoria: **Relajado** (el más débil), **Adquirir** (para lecturas también conocidas como cargas), **Liberar** (para escrituras también conocidas como tiendas), **AcqRel** (equivalente a "Adquirir para cargar y Liberar para almacenar"; útil cuando ambos están involucrados en una sola operación, como comparar y cambiar, y **SeqCst** (la más fuerte). En el siguiente ejemplo, demostraremos cómo los pedidos "relajados" difieren de los pedidos "Adquirir" y "Liberar".

```
use std::cell::UnsafeCell;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::{Arc, Barrier};
use std::thread;

struct UsizePair {
    atom: AtomicUsize,
    norm: UnsafeCell<usize>,
}

// UnsafeCell is not thread-safe. So manually mark our UsizePair to be Sync.
// (Effectively telling the compiler "I'll take care of it!")
unsafe impl Sync for UsizePair {}

static NTHREADS: usize = 8;
static NITERS: usize = 1000000;

fn main() {
    let upair = Arc::new(UsizePair::new(0));

    // Barrier is a counter-like synchronization structure (not to be confused
    // with a memory barrier). It blocks on a `wait` call until a fixed number
    // of `wait` calls are made from various threads (like waiting for all
    // players to get to the starting line before firing the starter pistol).
    let barrier = Arc::new(Barrier::new(NTHREADS + 1));

    let mut children = vec![];
```

```
    for _ in 0..NTHREADS {
        let upair = upair.clone();
        let barrier = barrier.clone();
        children.push(thread::spawn(move || {
            barrier.wait();

            let mut v = 0;
            while v < NITERS - 1 {
                // Read both members `atom` and `norm`, and check whether `atom`
                // contains a newer value than `norm`. See `UsizePair` impl for
                // details.
                let (atom, norm) = upair.get();
                if atom > norm {
                    // If `Acquire`-`Release` ordering is used in `get` and
                    // `set`, then this statement will never be reached.
                    println!("Reordered! {} > {}", atom, norm);
                }
                v = atom;
            }
        }));
    }

    barrier.wait();

    for v in 1..NITERS {
        // Update both members `atom` and `norm` to value `v`. See the impl for
        // details.
        upair.set(v);
    }

    for child in children {
        let _ = child.join();
    }
}

impl UsizePair {
    pub fn new(v: usize) -> UsizePair {
        UsizePair {
            atom: AtomicUsize::new(v),
            norm: UnsafeCell::new(v),
        }
    }

    pub fn get(&self) -> (usize, usize) {
        let atom = self.atom.load(Ordering::Relaxed); //Ordering::Acquire

        // If the above load operation is performed with `Acquire` ordering,
        // then all writes before the corresponding `Release` store is
        // guaranteed to be visible below.

        let norm = unsafe { *self.norm.get() };
        (atom, norm)
    }

    pub fn set(&self, v: usize) {
        unsafe { *self.norm.get() = v };

        // If the below store operation is performed with `Release` ordering,
        // then the write to `norm` above is guaranteed to be visible to all
        // threads that "loads `atom` with `Acquire` ordering and sees the same
        // value that was stored below". However, no guarantees are provided as
```

```
        // to when other readers will witness the below store, and consequently
        // the above write. On the other hand, there is also no guarantee that
        // these two values will be in sync for readers. Even if another thread
        // sees the same value that was stored below, it may actually see a
        // "later" value in `norm` than what was written above. That is, there
        // is no restriction on visibility into the future.

        self.atom.store(v, Ordering::Relaxed); //Ordering::Release
    }
}
```

Nota: las arquitecturas x86 tienen un fuerte modelo de memoria. Este post lo explica en detalle. También eche un vistazo a la página de Wikipedia para la comparación de arquitecturas.

Cerraduras de lectura-escritura

RwLocks permite que un solo productor proporcione datos a cualquier número de lectores mientras evita que los lectores vean datos no válidos o inconsistentes.

El siguiente ejemplo utiliza RwLock para mostrar cómo un solo subproceso productor puede aumentar periódicamente un valor mientras dos subprocesos de los consumidores leen el valor.

```
use std::time::Duration;
use std::thread;
use std::thread::sleep;
use std::sync::{Arc, RwLock };

fn main() {
    // Create an u32 with an inital value of 0
    let initial_value = 0u32;

    // Move the initial value into the read-write lock which is wrapped into an atomic
reference
    // counter in order to allow safe sharing.
    let rw_lock = Arc::new(RwLock::new(initial_value));

    // Create a clone for each thread
    let producer_lock = rw_lock.clone();
    let consumer_id_lock = rw_lock.clone();
    let consumer_square_lock = rw_lock.clone();

    let producer_thread = thread::spawn(move || {
        loop {
            // write() blocks this thread until write-exclusive access can be acquired and
retuns an
            // RAII guard upon completion
            if let Ok(mut write_guard) = producer_lock.write() {
                // the returned write_guard implements `Deref` giving us easy access to the
target value
                *write_guard += 1;

                println!("Updated value: {}", *write_guard);
            }

            // ^
            // |   when the RAII guard goes out of the scope, write access will be dropped,
allowing
```

```
            // +~  other threads access the lock

            sleep(Duration::from_millis(1000));
        }
    });

    // A reader thread that prints the current value to the screen
    let consumer_id_thread = thread::spawn(move || {
        loop {
            // read() will only block when `producer_thread` is holding a write lock
            if let Ok(read_guard) = consumer_id_lock.read() {
                // the returned read_guard also implements `Deref`
                println!("Read value: {}", *read_guard);
            }

            sleep(Duration::from_millis(500));
        }
    });

    // A second reader thread is printing the squared value to the screen. Note that readers
don't
    // block each other so `consumer_square_thread` can run simultaneously with
`consumer_id_lock`.
    let consumer_square_thread = thread::spawn(move || {
        loop {
            if let Ok(lock) = consumer_square_lock.read() {
                let value = *lock;
                println!("Read value squared: {}", value * value);
            }

            sleep(Duration::from_millis(750));
        }
    });

    let _ = producer_thread.join();
    let _ = consumer_id_thread.join();
    let _ = consumer_square_thread.join();
}
```

Ejemplo de salida:

```
Updated value: 1
Read value: 1
Read value squared: 1
Read value: 1
Read value squared: 1
Updated value: 2
Read value: 2
Read value: 2
Read value squared: 4
Updated value: 3
Read value: 3
Read value squared: 9
Read value: 3
Updated value: 4
Read value: 4
Read value squared: 16
Read value: 4
Read value squared: 16
Updated value: 5
```

```
Read value: 5
Read value: 5
Read value squared: 25
...(Interrupted)...
```

Lea Paralelismo en línea: https://riptutorial.com/es/rust/topic/1222/paralelismo

Capítulo 35: Pautas inseguras

Introducción

Explique por qué ciertas cosas están marcadas como `unsafe` en Rust y por qué es posible que necesitemos usar esta escotilla de escape en ciertas situaciones (raras).

Examples

Carreras de datos

Las carreras de datos ocurren cuando una parte de la memoria es actualizada por una parte, mientras que otra intenta leerla o actualizarla simultáneamente (sin sincronización entre las dos). Veamos el ejemplo clásico de una carrera de datos utilizando un contador compartido.

```
use std::cell::UnsafeCell;
use std::sync::Arc;
use std::thread;

// `UnsafeCell` is a zero-cost wrapper which informs the compiler that "what it
// contains might be shared mutably." This is used only for static analysis, and
// gets optimized away in release builds.
struct RacyUsize(UnsafeCell<usize>);

// Since UnsafeCell is not thread-safe, the compiler will not auto-impl Sync for
// any type containig it. And manually impl-ing Sync is "unsafe".
unsafe impl Sync for RacyUsize {}

impl RacyUsize {
    fn new(v: usize) -> RacyUsize {
        RacyUsize(UnsafeCell::new(v))
    }

    fn get(&self) -> usize {
        // UnsafeCell::get() returns a raw pointer to the value it contains
        // Dereferencing a raw pointer is also "unsafe"
        unsafe { *self.0.get() }
    }

    fn set(&self, v: usize) { // note: `&self` and not `&mut self`
        unsafe { *self.0.get() = v }
    }
}

fn main() {
    let racy_num = Arc::new(RacyUsize::new(0));

    let mut handlers = vec![];
    for _ in 0..10 {
        let racy_num = racy_num.clone();
        handlers.push(thread::spawn(move || {
            for i in 0..1000 {
                if i % 200 == 0 {
```

```
                    // give up the time slice to scheduler
                    thread::yield_now();
                    // this is needed to interleave the threads so as to observe
                    // data race, otherwise the threads will most likely be
                    // scheduled one after another.
                }

                // increment by one
                racy_num.set(racy_num.get() + 1);
            }
        }));
    }

    for th in handlers {
        th.join().unwrap();
    }

    println!("{}", racy_num.get());
}
```

La salida será casi siempre menor que `10000` (10 subprocesos × 1000) cuando se ejecute en un procesador de múltiples núcleos.

En este ejemplo, una carrera de datos ha producido un valor lógicamente incorrecto pero aún significativo. Esto se debe a que solo una palabra estuvo involucrada en la carrera y, por lo tanto, una actualización no pudo haberla cambiado parcialmente. Pero las carreras de datos en general pueden producir valores corruptos que no son válidos para un tipo (tipo inseguro) cuando el objeto que se está compitiendo abarca varias palabras y / o produce valores que apuntan a ubicaciones de memoria no válidas (memoria insegura) cuando están involucrados punteros.

Sin embargo, el uso cuidadoso de primitivas atómicas puede permitir la construcción de estructuras de datos muy eficientes que pueden necesitar internamente realizar algunas de estas operaciones "inseguras" para realizar acciones que no son verificables estáticamente por el sistema de tipo de Rust, pero que son correctas en general (es decir, para construir una caja fuerte). abstracción).

Lea Pautas inseguras en línea: https://riptutorial.com/es/rust/topic/6018/pautas-inseguras

Capítulo 36: PhantomData

Examples

Usando PhantomData como un marcador de tipo

El uso del tipo `PhantomData` como este le permite usar un tipo específico sin necesidad de que sea parte de Struct.

```
use std::marker::PhantomData;

struct Authenticator<T: GetInstance> {
    _marker: PhantomData<*const T>, // Using `*const T` indicates that we do not own a T
}

impl<T: GetInstance> Authenticator<T> {
    fn new() -> Authenticator<T> {
        Authenticator {
            _marker: PhantomData,
        }
    }

    fn auth(&self, id: i64) -> bool {
        T::get_instance(id).is_some()
    }
}

trait GetInstance {
    type Output; // Using nightly this could be defaulted to `Self`
    fn get_instance(id: i64) -> Option<Self::Output>;
}

struct Foo;

impl GetInstance for Foo {
    type Output = Self;
    fn get_instance(id: i64) -> Option<Foo> {
        // Here you could do something like a Database lookup or similarly
        if id == 1 {
            Some(Foo)
        } else {
            None
        }
    }
}

struct User;

impl GetInstance for User {
    type Output = Self;
    fn get_instance(id: i64) -> Option<User> {
        // Here you could do something like a Database lookup or similarly
        if id == 2 {
            Some(User)
        } else {
            None
```

```
        }
    }
}

fn main() {
    let user_auth = Authenticator::<User>::new();
    let other_auth = Authenticator::<Foo>::new();

    assert!(user_auth.auth(2));
    assert!(!user_auth.auth(1));

    assert!(other_auth.auth(1));
    assert!(!other_auth.auth(2));

}
```

Lea PhantomData en línea: https://riptutorial.com/es/rust/topic/7226/phantomdata

Capítulo 37: Propiedad

Introducción

La propiedad es uno de los conceptos más importantes en Rust, y es algo que no está presente en la mayoría de los otros idiomas. La idea de que un valor puede ser *propiedad* de una variable en particular es a menudo bastante difícil de entender, especialmente en los idiomas donde la copia está implícita, pero esta sección revisará las diferentes ideas que rodean la propiedad.

Sintaxis

- let x: & T = ... // x es una referencia inmutable
- let x: & mut T = ... // x es una referencia exclusiva y mutable
- vamos a _ = & mut foo; // pedir prestado foo de manera mudable (es decir, exclusivamente)
- deja _ = & foo; // tomar prestado foo de manera inmutable
- dejar _ = foo; // mover foo (requiere propiedad)

Observaciones

- En versiones mucho más antiguas de Rust (antes de 1.0; mayo de 2015), una variable propiedad tenía un tipo que comenzaba con `~` . Puedes ver esto en ejemplos muy antiguos.

Examples

Propiedad y préstamo

Todos los valores en Rust tienen exactamente un propietario. El propietario es responsable de eliminar ese valor cuando está fuera del alcance, y es el único que puede *mover* la propiedad del valor. El propietario de un valor puede regalar *referencias* al permitir que otras piezas de código *tomen* ese valor *prestado* . En cualquier momento, puede haber cualquier número de referencias inmutables a un valor:

```
let owned = String::from("hello");
// since we own the value, we may let other variables borrow it
let immutable_borrow1 = &owned;
// as all current borrows are immutable, we can allow many of them
let immutable_borrow2 = &owned;
// in fact, if we have an immutable reference, we are also free to
// duplicate that reference, since we maintain the invariant that
// there are only immutable references
let immutable_borrow3 = &*immutable_borrow2;
```

o una sola referencia mutable (`ERROR` denota un error en tiempo de compilación):

```
// for us to borrow a value mutably, it must be mutable
let mut owned = String::from("hello");
```

```
// we can borrow owned mutably
let mutable_borrow = &mut owned;
// but note that we cannot borrow owned *again*
let mutable_borrow2 = &mut owned; // ERROR, already borrowed
// nor can we cannot borrow owned immutably
// since a mutable borrow is exclusive.
let immutable_borrow = &owned; // ERROR, already borrowed
```

Si hay referencias pendientes (mutables o inmutables) a un valor, ese valor no se puede mover (es decir, su propiedad ha sido regalada). Tendríamos que asegurarnos de que todas las referencias se eliminaron primero para poder mover un valor:

```
let foo = owned; // ERROR, outstanding references to owned
let owned = String::from("hello");
{
    let borrow = &owned;
    // ...
} // the scope ends the borrow
let foo = owned; // OK, owned and not borrowed
```

Préstamos y vidas

Todos los valores en Rust tienen *toda* una *vida* . La vida útil de un valor abarca el segmento de código desde el valor que se introduce hasta donde se mueve, o el final del ámbito de contenido.

```
{
    let x = String::from("hello"); //             +
    // ...                                       :
    let y = String::from("hello"); //      +      |
    // ...                                 :      :
    foo(x) // x is moved                   |      = x's lifetime
    // ...                                 :
} //                                       = y's lifetime
```

Cada vez que toma prestado un valor, la referencia resultante tiene una *duración* que está ligada a la duración del valor prestado:

```
{
    let x = String::from("hello");
    let y = String::from("world");
    // when we borrow y here, the lifetime of the reference
    // stored in foo is equal to the lifetime of y
    // (i.e., between let y = above, to the end of the scope below)
    let foo = &y;
    // similarly, this reference to x is bound to the lifetime
    // of x --- bar cannot, for example, spawn a thread that uses
    // the reference beyond where x is moved below.
    bar(&x);
}
```

Propiedad y llamadas a funciones.

La mayoría de las preguntas sobre la propiedad surgen al escribir funciones. Cuando especifique

los tipos de argumentos de una función, puede elegir *cómo* se pasa ese valor. Si solo necesita acceso de solo lectura, puede tomar una referencia inmutable:

```
fn foo(x: &String) {
    // foo is only authorized to read x's contents, and to create
    // additional immutable references to it if it so desires.
    let y = *x; // ERROR, cannot move when not owned
    x.push_str("foo"); // ERROR, cannot mutate with immutable reference
    println!("{}", x.len()); // reading OK
    foo(x); // forwarding reference OK
}
```

Si `foo` necesita modificar el argumento, debe tomar una referencia exclusiva y mutable:

```
fn foo(x: &mut String) {
    // foo is still not responsible for dropping x before returning,
    // nor is it allowed to. however, foo may modify the String.
    let x2 = *x; // ERROR, cannot move when not owned
    x.push_str("foo"); // mutating OK
    drop(*x); // ERROR, cannot drop value when not owned
    println!("{}", x.len()); // reading OK
}
```

Si no especifica ya sea `&` o `&mut` , está diciendo que la función tomará posesión de un argumento. Esto significa que `foo` ahora también es responsable de soltar `x` .

```
fn foo(x: String) {
    // foo may do whatever it wishes with x, since no-one else has
    // access to it. once the function terminates, x will be dropped,
    // unless it is moved away when calling another function.
    let mut x2 = x; // moving OK
    x2.push_str("foo"); // mutating OK
    let _ = &mut x2; // mutable borrow OK
    let _ = &x2; // immutable borrow OK (note that &mut above is dropped)
    println!("{}", x2.len()); // reading OK
    drop(x2); // dropping OK
}
```

La propiedad y el rasgo de la copia

Algunos tipos de Rust implementan el rasgo de `Copy` . Los tipos que son `Copy` se pueden mover sin poseer el valor en cuestión. Esto se debe a que el contenido del valor puede simplemente copiarse byte por byte en la memoria para producir un nuevo valor idéntico. La mayoría de las primitivas en Rust (`bool` , `usize` , `f64` , etc.) son `Copy` .

```
let x: isize = 42;
let xr = &x;
let y = *xr; // OK, because isize is Copy
// both x and y are owned here
```

En particular, `Vec` y `String` *no* son `Copy` :

```
let x = Vec::new();
```

```
let xr = &x;
let y = *xr; // ERROR, cannot move out of borrowed content
```

Lea Propiedad en línea: https://riptutorial.com/es/rust/topic/4395/propiedad

Capítulo 38: Pruebas

Examples

Probar una función

```
fn to_test(output: bool) -> bool {
    output
}

#[cfg(test)] // The module is only compiled when testing.
mod test {
    use super::to_test;

    // This function is a test function. It will be executed and
    // the test will succeed if the function exits cleanly.
    #[test]
    fn test_to_test_ok() {
        assert_eq!(to_test(true), true);
    }

    // That test on the other hand will only succeed when the function
    // panics.
    #[test]
    #[should_panic]
    fn test_to_test_fail() {
        assert_eq!(to_test(true), false);
    }
}
```

(Enlace de juegos)

Ejecutar con `cargo test` .

Pruebas de integración

`lib.rs` :

```
pub fn to_test(output: bool) -> bool {
    output
}
```

Cada archivo en las `tests/` carpeta se compila como caja única. `tests/integration_test.rs`

```
extern crate test_lib;
use test_lib::to_test;

#[test]
fn test_to_test(){
    assert_eq!(to_test(true), true);
}
```

Pruebas de referencia

Con las pruebas de referencia puede probar y medir la velocidad del código, sin embargo, las pruebas de referencia siguen siendo inestables. Para habilitar los puntos de referencia en su proyecto de carga que necesita óxido nocturno, coloque sus pruebas de punto de referencia de integración en los `benches/` carpetas en la raíz de su proyecto de Carga, y ejecute el `cargo bench` .

Ejemplos de llogiq.github.io

```
extern crate test;
extern crate rand;

use test::Bencher;
use rand::Rng;
use std::mem::replace;

#[bench]
fn empty(b: &mut Bencher) {
    b.iter(|| 1)
}

#[bench]
fn setup_random_hashmap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::HashMap::new();

    b.iter(|| { map.insert(rng.gen::<u8>() as usize, val); val += 1; })
}

#[bench]
fn setup_random_vecmap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::VecMap::new();

    b.iter(|| { map.insert((rng.gen::<u8>()) as usize, val); val += 1; })
}

#[bench]
fn setup_random_vecmap_cap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::VecMap::with_capacity(256);

    b.iter(|| { map.insert((rng.gen::<u8>()) as usize, val); val += 1; })
}
```

Lea Pruebas en línea: https://riptutorial.com/es/rust/topic/961/pruebas

Capítulo 39: Punteros en bruto

Sintaxis

- deje que raw_ptr = & pointee como * const *type* // cree un puntero en bruto constante a algunos datos
- let raw_mut_ptr = & mut pointee como * mut *type* // crea un puntero raw mutable a algunos datos mutables
- Deje que deref = * raw_ptr // desreferencia un puntero en bruto (requiere un bloque no seguro)

Observaciones

- No se garantiza que los punteros sin procesar apunten a una dirección de memoria válida y, como tal, el uso descuidado puede dar lugar a errores inesperados (y probablemente fatales).
- Cualquier referencia de Rust normal (por ejemplo, `&my_object` donde el tipo de `my_object` es T) `my_object` a `*const T` Del mismo modo, las referencias mutables obligan a `*mut T`
- Los punteros sin procesar no mueven la propiedad (en contraste con los valores de Cuadro que)

Examples

Creando y utilizando punteros crudos constantes.

```
// Let's take an arbitrary piece of data, a 4-byte integer in this case
let some_data: u32 = 14;

// Create a constant raw pointer pointing to the data above
let data_ptr: *const u32 = &some_data as *const u32;

// Note: creating a raw pointer is totally safe but dereferencing a raw pointer requires an
// unsafe block
unsafe {
    let deref_data: u32 = *data_ptr;
    println!("Dereferenced data: {}", deref_data);
}
```

Se emitirá el código anterior: `Dereferenced data: 14`

Creando y utilizando punteros en bruto mutables.

```
// Let's take a mutable piece of data, a 4-byte integer in this case
let mut some_data: u32 = 14;

// Create a mutable raw pointer pointing to the data above
let data_ptr: *mut u32 = &mut some_data as *mut u32;
```

```
// Note: creating a raw pointer is totally safe but dereferencing a raw pointer requires an
// unsafe block
unsafe {
    *data_ptr = 20;
    println!("Dereferenced data: {}", some_data);
}
```

Se emitirá el código anterior: `Dereferenced data: 20`

Inicializando un puntero crudo a nulo

A diferencia de las referencias de Rust normales, los punteros sin formato pueden tomar valores nulos.

```
use std::ptr;

// Create a const NULL pointer
let null_ptr: *const u16 = ptr::null();

// Create a mutable NULL pointer
let mut_null_ptr: *mut u16 = ptr::null_mut();
```

Cadena de desreferenciación

Al igual que en C, los punteros en bruto Rust pueden apuntar a otros punteros en bruto (que a su vez pueden apuntar a otros punteros en bruto).

```
// Take a regular string slice
let planet: &str = "Earth";

// Create a constant pointer pointing to our string slice
let planet_ptr: *const &str = &planet as *const &str;

// Create a constant pointer pointing to the pointer
let planet_ptr_ptr: *const *const &str = &planet_ptr as *const *const &str;

// This can go on...
let planet_ptr_ptr_ptr = &planet_ptr_ptr as *const *const *const &str;

unsafe {
    // Direct usage
    println!("The name of our planet is: {}", planet);
    // Single dereference
    println!("The name of our planet is: {}", *planet_ptr);
    // Double dereference
    println!("The name of our planet is: {}", **planet_ptr_ptr);
    // Triple dereference
    println!("The name of our planet is: {}", ***planet_ptr_ptr_ptr);
}
```

Esto dará como resultado: `The name of our planet is: Earth` cuatro veces.

Mostrando punteros en bruto

Rust tiene un formateador predeterminado para los tipos de punteros que se pueden usar para mostrar punteros.

```
use std::ptr;

// Create some data, a raw pointer pointing to it and a null pointer
let data: u32 = 42;
let raw_ptr = &data as *const u32;
let null_ptr = ptr::null() as *const u32;

// the {:p} mapping shows pointer values as hexadecimal memory addresses
println!("Data address: {:p}", &data);
println!("Raw pointer address: {:p}", raw_ptr);
println!("Null pointer address: {:p}", null_ptr);
```

Esto producirá algo como esto:

```
Data address: 0x7fff59f6bcc0
Raw pointer address: 0x7fff59f6bcc0
Null pointer address: 0x0
```

Lea Punteros en bruto en línea: https://riptutorial.com/es/rust/topic/7270/punteros-en-bruto

Capítulo 40: Rasgos

Introducción

Los rasgos son una forma de describir un "contrato" que una `struct` debe implementar. Los rasgos generalmente definen las firmas de los métodos, pero también pueden proporcionar implementaciones basadas en otros métodos del rasgo, siempre que los *límites de* los *rasgos lo* permitan.

Para aquellos familiarizados con la programación orientada a objetos, los rasgos se pueden considerar como interfaces con algunas diferencias sutiles.

Sintaxis

- rasgo Rasgo {método fn (...) -> ReturnType; ...}
- rasgo Rasgo: Bound {método fn (...) -> ReturnType; ...}
- impl Trait for Type {fn method (...) -> ReturnType {...} ...}
- impl <T> Rasgo para T donde T: Límites {método fn (...) -> ReturnType {...} ...}

Observaciones

- Los rasgos se suelen comparar con las interfaces, pero es importante hacer una distinción entre los dos. En lenguajes OO como Java, las interfaces son una parte integral de las clases que las extienden. En Rust, el compilador no sabe nada de los rasgos de una estructura a menos que se usen esos rasgos.

Examples

Lo esencial

Creando un Rasgo

```
trait Speak {
    fn speak(&self) -> String;
}
```

Implementando un Rasgo

```
struct Person;
struct Dog;

impl Speak for Person {
    fn speak(&self) -> String {
        String::from("Hello.")
```

```
    }
}

impl Speak for Dog {
    fn speak(&self) -> String {
        String::from("Woof.")
    }
}

fn main() {
    let person = Person {};
    let dog = Dog {};
    println!("The person says {}", person.speak());
    println!("The dog says {}", dog.speak());
}
```

Despacho estático y dinámico

Es posible crear una función que acepte objetos que implementen un rasgo específico.

Despacho estático

```
fn generic_speak<T: Speak>(speaker: &T) {
    println!("{0}", speaker.speak());
}

fn main() {
    let person = Person {};
    let dog = Dog {};

    generic_speak(&person);
    generic_speak(&dog);
}
```

Aquí se usa el envío estático, lo que significa que el compilador Rust generará versiones especializadas de la función `generic_speak` para los tipos `Dog` y `Person` . Esta generación de versiones especializadas de una función polimórfica (o cualquier entidad polimórfica) durante la compilación se denomina **monomorfización** .

Despacho dinámico

```
fn generic_speak(speaker: &Speak) {
    println!("{0}", speaker.speak());
}

fn main() {
    let person = Person {};
    let dog = Dog {};

    generic_speak(&person as &Speak);
    generic_speak(&dog); // gets automatically coerced to &Speak
}
```

Aquí, solo existe una versión única de `generic_speak` en el binario compilado, y la llamada `speak()` se realiza mediante una búsqueda vtable en tiempo de ejecución. Por lo tanto, el uso del envío dinámico da como resultado una compilación más rápida y un tamaño más pequeño del binario compilado, mientras que es un poco más lento en el tiempo de ejecución.

Los objetos de tipo `&Speak` o `Box<Speak>` se denominan **objetos de rasgo** .

Tipos asociados

- Utilice el tipo asociado cuando exista una relación de uno a uno entre el tipo que implementa el rasgo y el tipo asociado.
- A veces también se conoce como el *tipo de salida* , ya que este es un elemento dado a un tipo cuando le aplicamos un rasgo.

Creación

```
trait GetItems {
    type First;
//  ^~~~ defines an associated type.
    type Last: ?Sized;
//           ^~~~~~~~ associated types may be constrained by traits as well
    fn first_item(&self) -> &Self::First;
//                           ^~~~~~~~~~~ use `Self::` to refer to the associated type
    fn last_item(&self) -> &Self::Last;
//                          ^~~~~~~~~~ associated types can be used as function output...
    fn set_first_item(&mut self, item: Self::First);
//                                     ^~~~~~~~~~~  ... input, and anywhere.
}
```

Implementacion

```
impl<T, U: ?Sized> GetItems for (T, U) {
    type First = T;
    type Last = U;
//              ^~~ assign the associated types
    fn first_item(&self) -> &Self::First { &self.0 }
    fn last_item(&self) -> &Self::Last { &self.1 }
    fn set_first_item(&mut self, item: Self::First) { self.0 = item; }
}

impl<T> GetItems for [T; 3] {
    type First = T;
    type Last = T;
    fn first_item(&self) -> &T { &self[0] }
//                            ^ you could refer to the actual type instead of `Self::First`
    fn last_item(&self) -> &T { &self[2] }
    fn set_first_item(&mut self, item: T) { self[0] = item; }
}
```

Haciendo referencia a tipos asociados.

Si estamos seguros de que un tipo `T` implementa `GetItems` por ejemplo, en genéricos, simplemente podríamos usar `T::First` para obtener el tipo asociado.

```
fn get_first_and_last<T: GetItems>(obj: &T) -> (&T::First, &T::Last) {
//                                              ^~~~~~~~ refer to an associated type
    (obj.first_item(), obj.last_item())
}
```

De lo contrario, debe indicar explícitamente al compilador qué rasgo está implementando el tipo

```
let array: [u32; 3] = [1, 2, 3];
let first: &<[u32; 3] as GetItems>::First = array.first_item();
//          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~ [u32; 3] may implement multiple traits which many
//                                        of them provide the `First` associated type.
//                                        thus the explicit "cast" is necessary here.
assert_eq!(*first, 1);
```

Restricción con tipos asociados

```
fn clone_first_and_last<T: GetItems>(obj: &T) -> (T::First, T::Last)
    where T::First: Clone, T::Last: Clone
//  ^~~~ use the `where` clause to constraint associated types by traits
{
    (obj.first_item().clone(), obj.last_item().clone())
}

fn get_first_u32<T: GetItems<First=u32>>(obj: &T) -> u32 {
//                          ^~~~~~~~~~~ constraint associated types by equality
    *obj.first_item()
}
```

Métodos predeterminados

```
trait Speak {
    fn speak(&self) -> String {
        String::from("Hi.")
    }
}
```

El método se llamará de forma predeterminada, excepto si se sobrescribe en el bloque `impl` .

```
struct Human;
struct Cat;

impl Speak for Human {}

impl Speak for Cat {
    fn speak(&self) -> String {
```

```
        String::from("Meow.")
    }
}

fn main() {
    let human = Human {};
    let cat = Cat {};
    println!("The human says {}", human.speak());
    println!("The cat says {}", cat.speak());
}
```

Salida:

> El humano dice hola.
>
> El gato dice miau.

Poner un límite en un rasgo

Al definir un nuevo rasgo, es posible imponer que los tipos que deseen implementar este rasgo verifiquen una serie de restricciones o límites.

Tomando un ejemplo de la biblioteca estándar, el rasgo `DerefMut` requiere que un tipo implemente primero su rasgo `Deref` hermano:

```
pub trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

Esto, a su vez, permite que `DerefMut` use el tipo asociado `Target` definido por `Deref` .

Mientras que la sintaxis podría ser una reminiscencia de la herencia:

- trae todos los elementos asociados (constantes, tipos, funciones, ...) del rasgo enlazado
- permite el polimorfismo de `&DerefMut` a `&Deref`

Esto es diferente en la naturaleza:

- es posible usar un tiempo de vida (como `'static`) como un límite
- no es posible anular los elementos de rasgos vinculados (ni siquiera las funciones)

Por lo tanto, es mejor pensar en él como un concepto separado.

Múltiples tipos de objetos enlazados

También es posible agregar varios tipos de objetos a una función de envío estático .

```
fn mammal_speak<T: Person + Dog>(mammal: &T) {
    println!("{0}", mammal.speak());
}
```

```
fn main() {
    let person = Person {};
    let dog = Dog {};

    mammal_speak(&person);
    mammal_speak(&dog);
}
```

Lea Rasgos en línea: https://riptutorial.com/es/rust/topic/1313/rasgos

Capítulo 41: Rasgos de conversión

Observaciones

- `AsRef` y `Borrow` son similares pero tienen propósitos distintos. `Borrow` se utiliza para tratar métodos de endeudamiento múltiples de manera similar, o para tratar valores prestados como sus contrapartes de propiedad, mientras que `AsRef` se utiliza para generar referencias.
- `From<A> for B` implica `Into<B> for A` , pero no al revés.
- `From<A> for A` se implementa implícitamente.

Examples

Desde

El rasgo `From` de Rust es un rasgo de propósito general para la conversión entre tipos. Para cualquiera de los dos tipos `TypeA` y `TypeB` ,

```
impl From<TypeA> for TypeB
```

indica que se *garantiza* que una instancia de `TypeB` se puede construir desde una instancia de `TypeA` . Una implementación de `From` ve así:

```
struct TypeA {
    a: u32,
}

struct TypeB {
    b: u32,
}

impl From<TypeA> for TypeB {
    fn from(src: TypeA) -> Self {
        TypeB {
            b: src.a,
        }
    }
}
```

AsRef & AsMut

`std::convert::AsRef` y `std::convert::AsMut` se utilizan para convertir tipos de referencias a bajo costo. Para los tipos `A` y `B` ,

```
impl AsRef<B> for A
```

indica que a `&A` se puede convertir en a `&B` y,

```
impl AsMut<B> for A
```

indica que un `&mut A` se puede convertir en un `&mut B`

Esto es útil para realizar conversiones de tipo sin copiar o mover valores. Un ejemplo en la biblioteca estándar es `std::fs::File.open()` :

```
fn open<P: AsRef<Path>>(path: P) -> Result<File>
```

Esto permite que `File.open()` acepte no solo `Path` , sino también `OsStr` , `OsString` , `str` , `String` y `PathBuf` con conversión implícita porque todos estos tipos implementan `AsRef<Path>` .

Pedir prestado

Los rasgos `std::borrow::Borrow` y `std::borrow::BorrowMut` se utilizan para tratar tipos prestados como tipos de propiedad. Para los tipos `A` y `B` ,

```
impl Borrow<B> for A
```

indica que se puede usar una `A` prestada donde se desea una `B` Por ejemplo, `std::collections::HashMap.get()` usa `Borrow` para su método `get()` , lo que permite que un `HashMap` con claves de `A` se indexe con a `&B`

Por otro lado, `std::borrow::ToOwned` implementa la relación inversa.

Así, con los tipos `A` y `B` mencionados anteriormente se puede implementar:

```
impl ToOwned for B
```

Nota: mientras que `A` puede implementar `Borrow<T>` para varios tipos distintos `T` , `B` solo puede implementar `ToOwned` una vez.

Deref & DerefMut

Los rasgos `std::ops::Deref` y `std::ops::DerefMut` se utilizan para sobrecargar el operador de desreferencia, `*x` . Para los tipos `A` y `B` ,

```
impl Deref<Target=B> for A
```

indica que la anulación de la referencia de una unión de `&A` producirá a `&B` y,

```
impl DerefMut for A
```

indica que al anular una unión de `&mut A` obtendrá un `&mut B`

`Deref` (resp. `DerefMut`) también proporciona una función de lenguaje útil llamada *deref coercion* ,

que permite que `&A` (resp. `&mut A`) coaccione automáticamente a a `&B` (resp. `&mut B`). Esto se usa comúnmente cuando se convierte de `String` a `&str`, ya que `&String` se obliga implícitamente a `&str` según sea necesario.

Nota: `DerefMut` no admite la especificación del tipo resultante, usa el mismo tipo que `Deref`.

Nota: Debido al uso de un tipo asociado (a diferencia de `AsRef`), un tipo dado solo puede implementar cada uno de `Deref` y `DerefMut` una vez.

Lea Rasgos de conversión en línea: https://riptutorial.com/es/rust/topic/2661/rasgos-de-conversion

Capítulo 42: Redes TCP

Examples

Una aplicación simple de cliente y servidor TCP: echo

El siguiente código se basa en los ejemplos proporcionados por la documentación en std :: net :: TcpListener . Esta aplicación de servidor escuchará las solicitudes entrantes y devolverá todos los datos entrantes, actuando así como un servidor "eco". La aplicación cliente enviará un pequeño mensaje y esperará una respuesta con el mismo contenido.

servidor:

```
use std::thread;
use std::net::{TcpListener, TcpStream, Shutdown};
use std::io::{Read, Write};

fn handle_client(mut stream: TcpStream) {
    let mut data = [0 as u8; 50]; // using 50 byte buffer
    while match stream.read(&mut data) {
        Ok(size) => {
            // echo everything!
            stream.write(&data[0..size]).unwrap();
            true
        },
        Err(_) => {
            println!("An error occurred, terminating connection with {}",
stream.peer_addr().unwrap());
            stream.shutdown(Shutdown::Both).unwrap();
            false
        }
    } {}
}

fn main() {
    let listener = TcpListener::bind("0.0.0.0:3333").unwrap();
    // accept connections and process them, spawning a new thread for each one
    println!("Server listening on port 3333");
    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                println!("New connection: {}", stream.peer_addr().unwrap());
                thread::spawn(move|| {
                    // connection succeeded
                    handle_client(stream)
                });
            }
            Err(e) => {
                println!("Error: {}", e);
                /* connection failed */
            }
        }
    }
    // close the socket server
    drop(listener);
```

```
}
```

cliente:

```
use std::net::{TcpStream};
use std::io::{Read, Write};
use std::str::from_utf8;

fn main() {
    match TcpStream::connect("localhost:3333") {
        Ok(mut stream) => {
            println!("Successfully connected to server in port 3333");

            let msg = b"Hello!";

            stream.write(msg).unwrap();
            println!("Sent Hello, awaiting reply...");

            let mut data = [0 as u8; 6]; // using 6 byte buffer
            match stream.read_exact(&mut data) {
                Ok(_) => {
                    if &data == msg {
                        println!("Reply is ok!");
                    } else {
                        let text = from_utf8(&data).unwrap();
                        println!("Unexpected reply: {}", text);
                    }
                },
                Err(e) => {
                    println!("Failed to receive data: {}", e);
                }
            }
        },
        Err(e) => {
            println!("Failed to connect: {}", e);
        }
    }
    println!("Terminated.");
}
```

Lea Redes TCP en línea: https://riptutorial.com/es/rust/topic/1350/redes-tcp

Capítulo 43: Regex

Introducción

La biblioteca estándar de Rust no contiene ningún analizador / emparejador de expresiones regulares, pero la caja de expresiones `regex` (que está en el vivero de rust-lang y, por lo tanto, semioficial) proporciona un analizador de expresiones regulares. Esta sección de la documentación proporcionará una descripción general de cómo usar la caja `regex` en situaciones comunes, junto con las instrucciones de instalación y cualquier otra observación útil que se necesite al usar la caja.

Examples

Búsqueda simple

Soporte para expresiones regulares para tust es proporcionada por la `regex` cajón, agregarlo a su `Cargo.toml` :

```
[dependencies]
regex = "0.1"
```

La interfaz principal de la `regex` cajón es `regex::Regex` :

```
extern crate regex;
use regex::Regex;

fn main() {
    //"r" stands for "raw" strings, you probably
    // need them because rustc checks escape sequences,
    // although you can always use "\\" withour "r"
    let num_regex = Regex::new(r"\d+").unwrap();
    // is_match checks if string matches the pattern
    assert!(num_regex.is_match("some string with number 1"));

    let example_string = "some 123 numbers";
    // Regex::find searches for pattern and returns Option<(usize,usize)>,
    // which is either indexes of first and last bytes of match
    // or "None" if nothing matched
    match num_regex.find(example_string) {
        // Get the match slice from string, prints "123"
        Some(x) => println!("{}", &example_string[x.0 .. x.1]),
        None    => unreachable!()
    }
}
```

Grupos de captura

```
extern crate regex;
use regex::Regex;
```

```
fn main() {
    let rg = Regex::new(r"was (\d+)").unwrap();
    // Regex::captures returns Option<Captures>,
    // first element is the full match and others
    // are capture groups
    match rg.captures("The year was 2016") {
        // Access captures groups via Captures::at
        // Prints Some("2016")
        Some(x) => println!("{:?}", x.at(1)),
        None    => unreachable!()
    }

    // Regex::captures also supports named capture groups
    let rg_w_named = Regex::new(r"was (?P<year>\d+)").unwrap();
    match rg_w_named.captures("The year was 2016") {
        // Named capures groups are accessed via Captures::name
        // Prints Some("2016")
        Some(x) => println!("{:?}", x.name("year")),
        None    => unreachable!()
    }

}
```

Reemplazo

```
extern crate regex;
use regex::Regex;

fn main() {
    let rg = Regex::new(r"(\d+)").unwrap();

    // Regex::replace replaces first match
    // from it's first argument with the second argument
    // => Some string with numbers (not really)
    rg.replace("Some string with numbers 123", "(not really)");

    // Capture groups can be accesed via $number
    // => Some string with numbers (which are 123)
    rg.replace("Some string with numbers 123", "(which are $1)");

    let rg = Regex::new(r"(?P<num>\d+)").unwrap();

    // Named capture groups can be accesed via $name
    // => Some string with numbers (which are 123)
    rg.replace("Some string with numbers 123", "(which are $num)");

    // Regex::replace_all replaces all the matches, not only the first
    // => Some string with numbers (not really) (not really)
    rg.replace_all("Some string with numbers 123 321", "(not really)");
}
```

Lea Regex en línea: https://riptutorial.com/es/rust/topic/7184/regex

Capítulo 44: Rust orientado a objetos

Introducción

El óxido está orientado a los objetos porque sus tipos de datos algebraicos pueden tener métodos asociados, lo que los convierte en objetos en el sentido de datos almacenados junto con el código que sabe cómo trabajar con ellos.

Sin embargo, el óxido no admite la herencia, favoreciendo la composición con rasgos. Esto significa que muchos patrones OO no funcionan tal cual y deben modificarse. Algunos son totalmente irrelevantes.

Examples

Herencia con rasgos

En Rust, no existe el concepto de "heredar" las propiedades de una estructura. En cambio, cuando diseña la relación entre los objetos, hágalo de manera que la funcionalidad de cada uno esté definida por una interfaz (un **rasgo** en Rust). Esto promueve la composición sobre la herencia , que se considera más útil y más fácil de extender a proyectos más grandes.

Aquí hay un ejemplo usando alguna herencia de ejemplo en Python:

```
class Animal:
    def speak(self):
        print("The " + self.animal_type + " said " + self.noise)

class Dog(Animal):
    def __init__(self):
        self.animal_type = 'dog'
        self.noise = 'woof'
```

Para traducir esto a Rust, necesitamos sacar lo que constituye un animal y poner esa funcionalidad en rasgos.

```
trait Speaks {
     fn speak(&self);

     fn noise(&self) -> &str;
}

trait Animal {
    fn animal_type(&self) -> &str;
}

struct Dog {}

impl Animal for Dog {
    fn animal_type(&self) -> &str {
        "dog"
```

```
    }
}

impl Speaks for Dog {
    fn speak(&self) {
        println!("The dog said {}", self.noise());
    }

    fn noise(&self) -> &str {
        "woof"
    }
}

fn main() {
    let dog = Dog {};
    dog.speak();
}
```

Observe cómo dividimos esa clase padre abstracta en dos componentes separados: la parte que define la estructura como un animal y la parte que le permite hablar.

Los lectores astutos notarán que esto no es uno a uno, ya que cada implementador debe reimplementar la lógica para imprimir una cadena con el formato "El {animal} dijo {ruido}". Puede hacerlo con un ligero rediseño de la interfaz donde implementamos `Speak` for `Animal` :

```
trait Speaks {
     fn speak(&self);
}

trait Animal {
    fn animal_type(&self) -> &str;
    fn noise(&self) -> &str;
}

impl<T> Speaks for T where T: Animal {
    fn speak(&self) {
        println!("The {} said {}", self.animal_type(), self.noise());
    }
}

struct Dog {}
struct Cat {}

impl Animal for Dog {
    fn animal_type(&self) -> &str {
        "dog"
    }

    fn noise(&self) -> &str {
        "woof"
    }
}

impl Animal for Cat {
    fn animal_type(&self) -> &str {
        "cat"
    }

    fn noise(&self) -> &str {
```

```
        "meow"
    }
}

fn main() {
    let dog = Dog {};
    let cat = Cat {};
    dog.speak();
    cat.speak();
}
```

Observe que ahora el animal hace un ruido y habla simplemente que ahora tiene una implementación para cualquier cosa que sea un animal. Esto es mucho más flexible que la forma anterior y la herencia de Python. Por ejemplo, si desea agregar un ser `Human` que tenga un sonido diferente, podemos simplemente tener otra implementación de `speak` por algo `Human` :

```
trait Human {
    fn name(&self) -> &str;
    fn sentence(&self) -> &str;
}

struct Person {}

impl<T> Speaks for T where T: Human {
    fn speak(&self) {
        println!("{} said {}", self.name(), self.sentence());
    }
}
```

Patrón de visitante

El ejemplo típico de visitante en Java sería:

```
interface ShapeVisitor {
    void visit(Circle c);
    void visit(Rectangle r);
}

interface Shape {
    void accept(ShapeVisitor sv);
}

class Circle implements Shape {
    private Point center;
    private double radius;

    public Circle(Point center, double radius) {
        this.center = center;
        this.radius = radius;
    }

    public Point getCenter() { return center; }
    public double getRadius() { return radius; }

    @Override
    public void accept(ShapeVisitor sv) {
        sv.visit(this);
```

```
    }
}

class Rectangle implements Shape {
    private Point lowerLeftCorner;
    private Point upperRightCorner;

    public Rectangle(Point lowerLeftCorner, Point upperRightCorner) {
        this.lowerLeftCorner = lowerLeftCorner;
        this.upperRightCorner = upperRightCorner;
    }

    public double length() { ... }
    public double width() { ... }

    @Override
    public void accept(ShapeVisitor sv) {
        sv.visit(this);
    }
}

class AreaCalculator implements ShapeVisitor {
    private double area = 0.0;

    public double getArea() { return area; }

    public void visit(Circle c) {
        area = Math.PI * c.radius() * c.radius();
    }

    public void visit(Rectangle r) {
         area = r.length() * r.width();
    }
}

double computeArea(Shape s) {
    AreaCalculator ac = new AreaCalculator();
    s.accept(ac);
    return ac.getArea();
}
```

Esto se puede traducir fácilmente a Rust, de dos maneras.

La primera forma utiliza el polimorfismo en tiempo de ejecución:

```
trait ShapeVisitor {
    fn visit_circle(&mut self, c: &Circle);
    fn visit_rectangle(&mut self, r: &Rectangle);
}

trait Shape {
    fn accept(&self, sv: &mut ShapeVisitor);
}

struct Circle {
    center: Point,
    radius: f64,
}

struct Rectangle {
```

```
    lowerLeftCorner: Point,
    upperRightCorner: Point,
}

impl Shape for Circle {
    fn accept(&self, sv: &mut ShapeVisitor) {
        sv.visit_circle(self);
    }
}

impl Rectangle {
    fn length() -> double { ... }
    fn width() -> double { ... }
}

impl Shape for Rectangle {
    fn accept(&self, sv: &mut ShapeVisitor) {
        sv.visit_rectangle(self);
    }
}

fn computeArea(s: &Shape) -> f64 {
    struct AreaCalculator {
        area: f64,
    }

    impl ShapeVisitor for AreaCalculator {
        fn visit_circle(&mut self, c: &Circle) {
            self.area = std::f64::consts::PI * c.radius * c.radius;
        }
        fn visit_rectangle(&mut self, r: &Rectangle) {
            self.area = r.length() * r.width();
        }
    }

    let mut ac = AreaCalculator { area: 0.0 };
    s.accept(&mut ac);
    ac.area
}
```

La segunda forma utiliza el polimorfismo en tiempo de compilación, solo se muestran las diferencias aquí:

```
trait Shape {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V);
}

impl Shape for Circle {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V) {
        // same body
    }
}

impl Shape for Rectangle {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V) {
        // same body
    }
}

fn computeArea<S: Shape>(s: &S) -> f64 {
```

```
    // same body
}
```

Lea Rust orientado a objetos en línea: https://riptutorial.com/es/rust/topic/6737/rust-orientado-a-objetos

Capítulo 45: Serde

Introducción

Serde es un popular ialization **ser** y marco **de** serialización de Rust, que se utiliza para convertir *los datos en serie* (por ejemplo, JSON y XML) para estructuras de óxido y viceversa. Serde soporta muchos formatos, incluyendo: JSON, YAML, TOML, BSON, Pickle y XML.

Examples

Struct ◂▸ JSON

main.rs

```
extern crate serde;
extern crate serde_json;

// Import this crate to derive the Serialize and Deserialize traits.
#[macro_use] extern crate serde_derive;

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

    // Convert the Point to a packed JSON string. To convert it to
    // pretty JSON with indentation, use `to_string_pretty` instead.
    let serialized = serde_json::to_string(&point).unwrap();

    // Prints serialized = {"x":1,"y":2}
    println!("serialized = {}", serialized);

    // Convert the JSON string back to a Point.
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    // Prints deserialized = Point { x: 1, y: 2 }
    println!("deserialized = {:?}", deserialized);
}
```

Cargo.toml

```
[package]
name = "serde-example"
version = "0.1.0"
```

```
build = "build.rs"

[dependencies]
serde = "0.9"
serde_json = "0.9"
serde_derive = "0.9"
```

Serializar enumeración como cadena

```
extern crate serde;
extern crate serde_json;

macro_rules! enum_str {
    ($name:ident { $($variant:ident($str:expr), )* }) => {
        #[derive(Clone, Copy, Debug, Eq, PartialEq)]
        pub enum $name {
            $($variant,)*
        }

        impl ::serde::Serialize for $name {
            fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
                where S: ::serde::Serializer,
            {
                // Serialize the enum as a string.
                serializer.serialize_str(match *self {
                    $( $name::$variant => $str, )*
                })
            }
        }

        impl ::serde::Deserialize for $name {
            fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
                where D: ::serde::Deserializer,
            {
                struct Visitor;

                impl ::serde::de::Visitor for Visitor {
                    type Value = $name;

                    fn expecting(&self, formatter: &mut ::std::fmt::Formatter) ->
::std::fmt::Result {
                        write!(formatter, "a string for {}", stringify!($name))
                    }

                    fn visit_str<E>(self, value: &str) -> Result<$name, E>
                        where E: ::serde::de::Error,
                    {
                        match value {
                            $( $str => Ok($name::$variant), )*
                            _ => Err(E::invalid_value(::serde::de::Unexpected::Other(
                                &format!("unknown {} variant: {}", stringify!($name), value)
                            ), &self)),
                        }
                    }
                }

                // Deserialize the enum from a string.
                deserializer.deserialize_str(Visitor)
            }
```

```
        }
    }
}

enum_str!(LanguageCode {
    English("en"),
    Spanish("es"),
    Italian("it"),
    Japanese("ja"),
    Chinese("zh"),
});

fn main() {
    use LanguageCode::*;
    let languages = vec![English, Spanish, Italian, Japanese, Chinese];

    // Prints ["en","es","it","ja","zh"]
    println!("{}", serde_json::to_string(&languages).unwrap());

    let input = r#" "ja" "#;
    assert_eq!(Japanese, serde_json::from_str(input).unwrap());
}
```

Serializar campos como camelCase

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

#[derive(Serialize)]
struct Person {
    #[serde(rename="firstName")]
    first_name: String,
    #[serde(rename="lastName")]
    last_name: String,
}

fn main() {
    let person = Person {
        first_name: "Joel".to_string(),
        last_name: "Spolsky".to_string(),
    };

    let json = serde_json::to_string_pretty(&person).unwrap();

    // Prints:
    //
    //    {
    //      "firstName": "Joel",
    //      "lastName": "Spolsky"
    //    }
    println!("{}", json);
}
```

Valor predeterminado para el campo

```
extern crate serde;
```

```
extern crate serde_json;
#[macro_use] extern crate serde_derive;

#[derive(Deserialize, Debug)]
struct Request {
    // Use the result of a function as the default if "resource" is
    // not included in the input.
    #[serde(default="default_resource")]
    resource: String,

    // Use the type's implementation of std::default::Default if
    // "timeout" is not included in the input.
    #[serde(default)]
    timeout: Timeout,

    // Use a method from the type as the default if "priority" is not
    // included in the input. This may also be a trait method.
    #[serde(default="Priority::lowest")]
    priority: Priority,
}

fn default_resource() -> String {
    "/".to_string()
}

/// Timeout in seconds.
#[derive(Deserialize, Debug)]
struct Timeout(u32);
impl Default for Timeout {
    fn default() -> Self {
        Timeout(30)
    }
}

#[derive(Deserialize, Debug)]
enum Priority { ExtraHigh, High, Normal, Low, ExtraLow }
impl Priority {
    fn lowest() -> Self { Priority::ExtraLow }
}

fn main() {
    let json = r#"
        [
          {
            "resource": "/users"
          },
          {
            "timeout": 5,
            "priority": "High"
          }
        ]
    "#;

    let requests: Vec<Request> = serde_json::from_str(json).unwrap();

    // The first request has resource="/users", timeout=30, priority=ExtraLow
    println!("{:?}", requests[0]);

    // The second request has resource="/", timeout=5, priority=High
    println!("{:?}", requests[1]);
}
```

Saltar campo de serialización

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

use std::collections::BTreeMap as Map;

#[derive(Serialize)]
struct Resource {
    // Always serialized.
    name: String,

    // Never serialized.
    #[serde(skip_serializing)]
    hash: String,

    // Use a method to decide whether the field should be skipped.
    #[serde(skip_serializing_if="Map::is_empty")]
    metadata: Map<String, String>,
}

fn main() {
    let resources = vec![
        Resource {
            name: "Stack Overflow".to_string(),
            hash: "b6469c3f31653d281bbbfa6f94d60fea130abe38".to_string(),
            metadata: Map::new(),
        },
        Resource {
            name: "GitHub".to_string(),
            hash: "5cb7a0c47e53854cd00e1a968de5abce1c124601".to_string(),
            metadata: {
                let mut metadata = Map::new();
                metadata.insert("headquarters".to_string(),
                                "San Francisco".to_string());
                metadata
            },
        },
    ];

    let json = serde_json::to_string_pretty(&resources).unwrap();

    // Prints:
    //
    //    [
    //      {
    //        "name": "Stack Overflow"
    //      },
    //      {
    //        "name": "GitHub",
    //        "metadata": {
    //          "headquarters": "San Francisco"
    //        }
    //      }
    //    ]
    println!("{}", json);
}
```

Implementar Serialize para un tipo de mapa personalizado

```
impl<K, V> Serialize for MyMap<K, V>
    where K: Serialize,
          V: Serialize
{
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer
    {
        let mut state = serializer.serialize_map(Some(self.len()))?;
        for (k, v) in self {
            state.serialize_entry(k, v)?;
        }
        state.end()
    }
}
```

Implementar Deserialize para un tipo de mapa personalizado

```
// A Visitor is a type that holds methods that a Deserializer can drive
// depending on what is contained in the input data.
//
// In the case of a map we need generic type parameters K and V to be
// able to set the output type correctly, but don't require any state.
// This is an example of a "zero sized type" in Rust. The PhantomData
// keeps the compiler from complaining about unused generic type
// parameters.
struct MyMapVisitor<K, V> {
    marker: PhantomData<MyMap<K, V>>
}

impl<K, V> MyMapVisitor<K, V> {
    fn new() -> Self {
        MyMapVisitor {
            marker: PhantomData
        }
    }
}

// This is the trait that Deserializers are going to be driving. There
// is one method for each type of data that our type knows how to
// deserialize from. There are many other methods that are not
// implemented here, for example deserializing from integers or strings.
// By default those methods will return an error, which makes sense
// because we cannot deserialize a MyMap from an integer or string.
impl<K, V> de::Visitor for MyMapVisitor<K, V>
    where K: Deserialize,
          V: Deserialize
{
    // The type that our Visitor is going to produce.
    type Value = MyMap<K, V>;

    // Deserialize MyMap from an abstract "map" provided by the
    // Deserializer. The MapVisitor input is a callback provided by
    // the Deserializer to let us see each entry in the map.
    fn visit_map<M>(self, mut visitor: M) -> Result<Self::Value, M::Error>
        where M: de::MapVisitor
    {
```

```
        let mut values = MyMap::with_capacity(visitor.size_hint().0);

        // While there are entries remaining in the input, add them
        // into our map.
        while let Some((key, value)) = visitor.visit()? {
            values.insert(key, value);
        }

        Ok(values)
    }

    // As a convenience, provide a way to deserialize MyMap from
    // the abstract "unit" type. This corresponds to `null` in JSON.
    // If your JSON contains `null` for a field that is supposed to
    // be a MyMap, we interpret that as an empty map.
    fn visit_unit<E>(self) -> Result<Self::Value, E>
        where E: de::Error
    {
        Ok(MyMap::new())
    }

    // When an unexpected data type is encountered, this method will
    // be invoked to inform the user what is actually expected.
    fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        write!(formatter, "a map or `null`")
    }
}

// This is the trait that informs Serde how to deserialize MyMap.
impl<K, V> Deserialize for MyMap<K, V>
    where K: Deserialize,
          V: Deserialize
{
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        // Instantiate our Visitor and ask the Deserializer to drive
        // it over the input data, resulting in an instance of MyMap.
        deserializer.deserialize_map(MyMapVisitor::new())
    }
}
```

Procesa una matriz de valores sin almacenarlos en un Vec

Supongamos que tenemos una matriz de enteros y queremos calcular el valor máximo sin tener que mantener toda la matriz en la memoria a la vez. Este enfoque puede adaptarse para manejar una variedad de otras situaciones en las que los datos deben procesarse mientras se deserializan en lugar de después.

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;
use serde::{de, Deserialize, Deserializer};

use std::cmp;
use std::fmt;
use std::marker::PhantomData;
```

```
#[derive(Deserialize)]
struct Outer {
    id: String,

    // Deserialize this field by computing the maximum value of a sequence
    // (JSON array) of values.
    #[serde(deserialize_with = "deserialize_max")]
    // Despite the struct field being named `max_value`, it is going to come
    // from a JSON field called `values`.
    #[serde(rename(deserialize = "values"))]
    max_value: u64,
}

/// Deserialize the maximum of a sequence of values. The entire sequence
/// is not buffered into memory as it would be if we deserialize to Vec<T>
/// and then compute the maximum later.
///
/// This function is generic over T which can be any type that implements
/// Ord. Above, it is used with T=u64.
fn deserialize_max<T, D>(deserializer: D) -> Result<T, D::Error>
    where T: Deserialize + Ord,
          D: Deserializer
{
    struct MaxVisitor<T>(PhantomData<T>);

    impl<T> de::Visitor for MaxVisitor<T>
        where T: Deserialize + Ord
    {
        /// Return type of this visitor. This visitor computes the max of a
        /// sequence of values of type T, so the type of the maximum is T.
        type Value = T;

        fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
            write!(formatter, "a sequence of numbers")
        }

        fn visit_seq<V>(self, mut visitor: V) -> Result<T, V::Error>
            where V: de::SeqVisitor
        {
            // Start with max equal to the first value in the seq.
            let mut max = match visitor.visit()? {
                Some(value) => value,
                None => {
                    // Cannot take the maximum of an empty seq.
                    let msg = "no values in seq when looking for maximum";
                    return Err(de::Error::custom(msg));
                }
            };

            // Update the max while there are additional values.
            while let Some(value) = visitor.visit()? {
                max = cmp::max(max, value);
            }

            Ok(max)
        }
    }

    // Create the visitor and ask the deserializer to drive it. The
    // deserializer will call visitor.visit_seq if a seq is present in
    // the input data.
```

```
    let visitor = MaxVisitor(PhantomData);
    deserializer.deserialize_seq(visitor)
}

fn main() {
    let j = r#"
        {
          "id": "demo-deserialize-max",
          "values": [
            256,
            100,
            384,
            314,
            271
          ]
        }
    "#;

    let out: Outer = serde_json::from_str(j).unwrap();

    // Prints "max value: 384"
    println!("max value: {}", out.max_value);
}
```

Límites de tipo genérico manuscritos

Cuando se derivan implementaciones de `Serialize` y `Deserialize` para estructuras con parámetros de tipo genérico, la mayoría de las veces Serde puede inferir los límites de rasgos correctos sin la ayuda del programador. Utiliza varias heurísticas para adivinar el límite correcto, pero lo más importante es que pone un límite de `T: Serialize` en cada parámetro de tipo `T` que forma parte de un campo serializado y un límite de `T: Deserialize` en cada parámetro de tipo `T` que forma parte de un campo deserializado. Al igual que con la mayoría de las heurísticas, esto no siempre es correcto y Serde proporciona una escotilla de escape para reemplazar el límite generado automáticamente por uno escrito por el programador.

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

use serde::de::{self, Deserialize, Deserializer};

use std::fmt::Display;
use std::str::FromStr;

#[derive(Deserialize, Debug)]
struct Outer<'a, S, T: 'a + ?Sized> {
    // When deriving the Deserialize impl, Serde would want to generate a bound
    // `S: Deserialize` on the type of this field. But we are going to use the
    // type's `FromStr` impl instead of its `Deserialize` impl by going through
    // `deserialize_from_str`, so we override the automatically generated bound
    // by the one required for `deserialize_from_str`.
    #[serde(deserialize_with = "deserialize_from_str")]
    #[serde(bound(deserialize = "S: FromStr, S::Err: Display"))]
    s: S,

    // Here Serde would want to generate a bound `T: Deserialize`. That is a
    // stricter condition than is necessary. In fact, the `main` function below
```

```
    // uses T=str which does not implement Deserialize. We override the
    // automatically generated bound by a looser one.
    #[serde(bound(deserialize = "Ptr<'a, T>: Deserialize"))]
    ptr: Ptr<'a, T>,
}

/// Deserialize a type `S` by deserializing a string, then using the `FromStr`
/// impl of `S` to create the result. The generic type `S` is not required to
/// implement `Deserialize`.
fn deserialize_from_str<S, D>(deserializer: D) -> Result<S, D::Error>
    where S: FromStr,
          S::Err: Display,
          D: Deserializer
{
    let s: String = try!(Deserialize::deserialize(deserializer));
    S::from_str(&s).map_err(|e| de::Error::custom(e.to_string()))
}

/// A pointer to `T` which may or may not own the data. When deserializing we
/// always want to produce owned data.
#[derive(Debug)]
enum Ptr<'a, T: 'a + ?Sized> {
    Ref(&'a T),
    Owned(Box<T>),
}

impl<'a, T: 'a + ?Sized> Deserialize for Ptr<'a, T>
    where Box<T>: Deserialize
{
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        let box_t = try!(Deserialize::deserialize(deserializer));
        Ok(Ptr::Owned(box_t))
    }
}

fn main() {
    let j = r#"
        {
          "s": "1234567890",
          "ptr": "owned"
        }
    "#;

    let result: Outer<u64, str> = serde_json::from_str(j).unwrap();

    // result = Outer { s: 1234567890, ptr: Owned("owned") }
    println!("result = {:?}", result);
}
```

Implementar Serializar y Deserializar para un tipo en una caja diferente

La regla de coherencia de Rust requiere que el rasgo o el tipo para el que está implementando el rasgo se definan en la misma caja que la impl, por lo que no es posible implementar Serialize y Deserialize para un tipo en una caja diferente directamente. El patrón newtype y la coacción `Deref` proporcionan una manera de implementar Serializar y Deserializar para un tipo que se comporta de la misma manera que el que usted quería.

```
use serde::{Serialize, Serializer, Deserialize, Deserializer};
use std::ops::Deref;

// Pretend this module is from some other crate.
mod not_my_crate {
    pub struct Data { /* ... */ }
}

// This single-element tuple struct is called a newtype struct.
struct Data(not_my_crate::Data);

impl Serialize for Data {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer
    {
        // Any implementation of Serialize.
    }
}

impl Deserialize for Data {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        // Any implementation of Deserialize.
    }
}

// Enable `Deref` coercion.
impl Deref for Data {
    type Target = not_my_crate::Data;
    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

// Now `Data` can be used in ways that require it to implement
// Serialize and Deserialize.
#[derive(Serialize, Deserialize)]
struct Outer {
    id: u64,
    name: String,
    data: Data,
}
```

Lea Serde en línea: https://riptutorial.com/es/rust/topic/1170/serde

Capítulo 46: The Drop Rasgo - Destructores en Rust

Observaciones

Usar el Rasgo de caída no significa que se ejecutará cada vez. Si bien se ejecutará cuando salga de alcance o se desenrolle, puede que no siempre sea así, por ejemplo, cuando se llama a `mem::forget` .

Esto se debe a que un pánico mientras se desenrolla hace que el programa se cancele. También podría haber sido compilado con el `Abort on Panic` activado.

Para obtener más información, consulte el libro: https://doc.rust-lang.org/book/drop.html

Examples

Implementación de Drop simple

```
use std::ops::Drop;

struct Foo(usize);

impl Drop for Foo {
    fn drop(&mut self) {
        println!("I had a {}", self.0);
    }
}
```

Gota para la limpieza

```
use std::ops::Drop;

#[derive(Debug)]
struct Bar(i32);

impl Bar {
    fn get<'a>(&'a mut self) -> Foo<'a> {
        let temp = self.0; // Since we will also capture `self` we..
                           // ..will have to copy the value out first
        Foo(self, temp) // Let's take the i32
    }
}

struct Foo<'a>(&'a mut Bar, i32); // We specify that we want a mutable borrow..
                                  // ..so we can put it back later on

impl<'a> Drop for Foo<'a> {
    fn drop(&mut self) {
        if self.1 < 10 { // This is just an example, you could also just put..
                         // ..it back as is
```

```
            (self.0).0 = self.1;
        }
    }
}

fn main() {
    let mut b = Bar(0);
    println!("{:?}", b);
    {
        let mut a : Foo = b.get(); // `a` now holds a reference to `b`..
        a.1 = 2;                   // .. and will hold it until end of scope
    }                              // .. here

    println!("{:?}", b);
    {
        let mut a : Foo = b.get();
        a.1 = 20;
    }
    println!("{:?}", b);
}
```

Drop te permite crear diseños simples y a prueba de fallas.

Drop Logging para la depuración de la gestión de memoria en tiempo de ejecución

La administración de memoria en tiempo de ejecución con Rc puede ser muy útil, pero también puede ser difícil envolver la cabeza, especialmente si su código es muy complejo y una sola instancia es referenciada por decenas o incluso cientos de otros tipos en muchos ámbitos.

Escribiendo un rasgo de Drop que incluye `println!("Dropping StructName: {:?}", self);` puede ser inmensamente valioso para la depuración, ya que le permite ver con precisión cuándo se agotan las referencias sólidas a una instancia de estructura.

Lea The Drop Rasgo - Destructores en Rust en línea: https://riptutorial.com/es/rust/topic/7233/the-drop-rasgo---destructores-en-rust

Capítulo 47: Tipos de datos primitivos

Examples

Tipos escalares

Enteros

Firmado: `i8` , `i16` , `i32` , `i64` , `isize`

Sin firmar : `u8` , `u16` , `u32` , `u64` , `usize`

El tipo de un entero literal, digamos `45` , se deducirá automáticamente del contexto. Pero para forzarlo, agregamos un sufijo: `45u8` (sin espacio) se escribirá `u8` .

Nota: El tamaño de `isize` y `usize` depende de la arquitectura. En el arco de 32 bits, es de 32 bits, y en 64 bits, lo has adivinado!

Puntos Flotantes

`f32` y `f64` .

Si solo escribe `2.0` , es `f64` por defecto, a menos que la inferencia de tipo determine lo contrario.

Para forzar `f32` , defina la variable con el tipo `f32` o el sufijo literal: `2.0f32` .

Booleanos

`bool` , teniendo valores `true` y `false` .

Caracteres

`char` , con valores escritos como `'x'` . En las comillas simples, contenga un solo valor escalar Unicode, lo que significa que es válido tener un emoji en él. Aquí hay 3 ejemplos más: `'❤'` , `'\u{3f}'` , `'\u{1d160}'` .

Lea Tipos de datos primitivos en línea: https://riptutorial.com/es/rust/topic/8705/tipos-de-datos-primitivos

Capítulo 48: Tuplas

Introducción

La estructura de datos más trivial, después de un valor singular, es la tupla.

Sintaxis

- (A, B, C) // una tupla de tres (una tupla con tres elementos), cuyo primer elemento tiene tipo A, segundo tipo B y tercero tipo C
- (A, B) // una tupla doble, cuyos dos elementos tienen tipo A y B respectivamente
- (A,) // a una tupla (nótese el trailing `,`), que tiene sólo un único elemento de tipo A
- () // la tupla vacía, que es tanto un tipo, como el único elemento de ese tipo

Examples

Tipos de tuplas y valores de tuplas

Las tuplas de óxido, como en la mayoría de los otros idiomas, son listas de tamaño fijo cuyos elementos pueden ser de diferentes tipos.

```
// Tuples in Rust are comma-separated values or types enclosed in parentheses.
let _ = ("hello", 42, true);
// The type of a tuple value is a type tuple with the same number of elements.
// Each element in the type tuple is the type of the corresponding value element.
let _: (i32, bool) = (42, true);
// Tuples can have any number of elements, including one ..
let _: (bool,) = (true,);
// .. or even zero!
let _: () = ();
// this last type has only one possible value, the empty tuple ()
// this is also the type (and value) of the return value of functions
// that do not return a value ..
let _: () = println!("hello");
// .. or of expressions with no value.
let mut a = 0;
let _: () = if true { a += 1; };
```

Coincidencia de valores de tupla

Los programas de oxidación utilizan la concordancia de patrones extensivamente para deconstruir los valores, ya sea utilizando `match` , `if let` , o deconstruyendo patrones de `let` . Las tuplas se pueden deconstruir como se podría esperar usando el `match`

```
fn foo(x: (&str, isize, bool)) {
    match x {
        (_, 42, _) => println!("it's 42"),
        (_, _, false) => println!("it's not true"),
```

```
        _ => println!("it's something else"),
    }
}
```

o con `if let`

```
fn foo(x: (&str, isize, bool)) {
    if let (_, 42, _) = x {
        println!("it's 42");
    } else {
        println!("it's something else");
    }
}
```

También puede enlazar dentro de la tupla usando `let` -deconstruction

```
fn foo(x: (&str, isize, bool)) {
    let (_, n, _) = x;
    println!("the number is {}", n);
}
```

Mirando dentro de las tuplas

Para acceder directamente a los elementos de una tupla, puede usar el formato `.n` para acceder al elemento `n` -ésimo

```
let x = ("hello", 42, true);
assert_eq!(x.0, "hello");
assert_eq!(x.1, 42);
assert_eq!(x.2, true);
```

También puedes moverte parcialmente de una tupla

```
let x = (String::from("hello"), 42);
let (s, _) = x;
let (_, n) = x;
println!("{}, {}", s, n);
// the following would fail however, since x.0 has already been moved
// let foo = x.0;
```

Lo esencial

Una tupla es simplemente una concatenación de múltiples valores:

- de tipos posiblemente diferentes
- cuyo número y tipos se conocen estáticamente

Por ejemplo, `(1, "Hello")` es una tupla de 2 elementos compuesta por `i32` y a `&str` , y su tipo se denota como `(i32, &'static str)` de manera similar a su valor.

Para acceder a un elemento de una tupla, uno simplemente usa su índice:

```
let tuple =  (1, "Hello");
println!("First element: {}, second element: {}", tuple.0, tuple.1);
```

Debido a que la tupla está incorporada, también es posible utilizar la coincidencia de patrones en las tuplas:

```
match (1, "Hello") {
    (i, _) if i < 0 => println!("Negative integer: {}", i),
    (_, s) => println!("{} World", s),
}
```

Casos especiales

La tupla del elemento 0: `()` también se denomina *unidad* , *tipo de unidad* o *tipo singleton* y se usa para denotar la ausencia de valores significativos. Es el tipo de retorno predeterminado de las funciones (cuando `->` no se especifica). *Ver también: ¿Qué tipo es el "tipo ()" en Rust?* .

La tupla de 1 elemento: `(a,)` , con la coma al final, denota una tupla de 1 elemento. La forma sin una coma `(a)` se interpreta como una expresión entre paréntesis y se evalúa como solo `a` .

Y mientras estamos en eso, siempre se aceptan las comas finales: `(1, "Hello",)` .

Limitaciones

El lenguaje Rust de hoy no admite *variadics* , además de las tuplas. Por lo tanto, no es posible implementar simplemente un rasgo para todas las tuplas y, como resultado, los rasgos estándar solo se implementan para tuplas hasta un número limitado de elementos (hoy, hasta 12 incluidos). Las tuplas con más elementos son compatibles, pero no implementan los rasgos estándar (aunque puede implementar sus propios rasgos para ellos).

Esperamos que esta restricción se levante en el futuro.

Desembalaje de tuplas

```
// It's possible to unpack tuples to assign their inner values to variables
let tup = (0, 1, 2);
// Unpack the tuple into variables a, b, and c
let (a, b, c) = tup;

assert_eq!(a, 0);
assert_eq!(b, 1);

// This works for nested data structures and other complex data types
let complex = ((1, 2), 3, Some(0));

let (a, b, c) = complex;
let (aa, ab) = a;

assert_eq!(aa, 1);
assert_eq!(ab, 2);
```

Lea Tuplas en línea: https://riptutorial.com/es/rust/topic/3941/tuplas

Capítulo 49: Valores en caja

Introducción

Las cajas son una parte muy importante de Rust, y cada rustáceo debe saber qué son y cómo usarlas

Examples

Creando una caja

En Rust estable, creas un Box usando la función `Box::new` .

```
let boxed_int: Box<i32> = Box::new(1);
```

Usando valores en caja

Debido a que las Cajas implementan el `Deref<Target=T>` , puede usar valores `Deref<Target=T>` como el valor que contienen.

```
let boxed_vec = Box::new(vec![1, 2, 3]);
println!("{}", boxed_vec.get(0));
```

Si desea un patrón de coincidencia en un valor de caja, es posible que tenga que eliminar la referencia de la caja manualmente.

```
struct Point {
    x: i32,
    y: i32,
}

let boxed_point = Box::new(Point { x: 0, y: 0});
// Notice the *. That dereferences the boxed value into just the value
match *boxed_point {
    Point {x, y} => println!("Point is at ({}, {})", x, y),
}
```

Uso de cajas para crear enums y estructuras recursivas

Si intenta crear una enumeración recursiva en Rust sin usar Box, obtendrá un error de tiempo de compilación que indica que la enumeración no se puede dimensionar.

```
// This gives an error!
enum List {
    Nil,
    Cons(i32, List)
}
```

Para que la enumeración tenga un tamaño definido, el valor contenido recursivamente debe estar en una casilla.

```
// This works!
enum List {
    Nil,
    Cons(i32, Box<List>)
}
```

Esto funciona porque Box siempre tiene el mismo tamaño, sin importar lo que sea T, lo que permite a Rust dar un tamaño a List.

Lea Valores en caja en línea: https://riptutorial.com/es/rust/topic/9341/valores-en-caja

Capítulo 50: Vidas

Sintaxis

- función fn <'a> (x: &' a Type)
- estructura Struct <'a> {x: &' a Type}
- enum Enum <'a> {Variant (&' a Type)}
- impl <'a> Struct <' a> {fn x <'a> (& self) -> &' a Type {self.x}}
- impl <'a> Rasgo <' a> para Tipo
- impl <'a> Rasgo para el Tipo <' a>
- `fn function<F>(f: F) where for<'a> F: FnOnce(&'a Type)`
- `struct Struct<F> where for<'a> F: FnOnce(&'a Type) { x: F }`
- `enum Enum<F> where for<'a> F: FnOnce(&'a Type) { Variant(F) }`
- `impl<F> Struct<F> where for<'a> F: FnOnce(&'a Type) { fn x(&self) -> &F { &self.x } }`

Observaciones

- Todas las referencias en Rust tienen una vida útil, incluso si no están anotadas explícitamente. El compilador es capaz de asignar implícitamente tiempos de vida.
- El `'static` tiempo de vida `'static` se asigna a las referencias que están almacenadas en el programa binario y serán válidas durante toda su ejecución. Esta vida útil se asigna principalmente a los literales de cadena, que tienen el tipo `&'static str` .

Examples

Parámetros de función (tiempos de vida de entrada)

```
fn foo<'a>(x: &'a u32) {
    // ...
}
```

Esto especifica que `foo` tiene una vida útil `'a` , y el parámetro `x` debe tener una vida útil de al menos `'a` . Las vidas útiles de las funciones generalmente se omiten a través de la *elision de* por *vida* :

```
fn foo(x: &u32) {
    // ...
}
```

En el caso de que una función tome múltiples referencias como parámetros y devuelva una referencia, el compilador no puede inferir el tiempo de vida del resultado a través de la *elision de* por *vida* .

```
error[E0106]: missing lifetime specifier
1 | fn foo(bar: &str, baz: &str) -> &i32 {
  |                                 ^ expected lifetime parameter
```

En su lugar, los parámetros de duración deben especificarse explícitamente.

```
// Return value of `foo` is valid as long as `bar` and `baz` are alive.
fn foo<'a>(bar: &'a str, baz: &'a str) -> &'a i32 {
```

Las funciones pueden tomar múltiples parámetros de por vida también.

```
// Return value is valid for the scope of `bar`
fn foo<'a, 'b>(bar: &'a str, baz: &'b str) -> &'a i32 {
```

Campos de fuerza

```
struct Struct<'a> {
    x: &'a u32,
}
```

Esto especifica que cualquier instancia dada de `Struct` tiene una vida útil `'a` , y el `&u32` almacenado en `x` debe tener una vida útil de al menos `'a` .

Bloques Impl

```
impl<'a> Type<'a> {
    fn my_function(&self) -> &'a u32 {
        self.x
    }
}
```

Esto especifica que el `Type` tiene una vida útil `'a` , y que la referencia devuelta por `my_function()` puede que ya no sea válida después de que `'a` termine porque el `Type` ya no existe para mantener `self.x`

Límites de rasgo de rango superior

```
fn copy_if<F>(slice: &[i32], pred: F) -> Vec<i32>
    where for<'a> F: Fn(&'a i32) -> bool
{
    let mut result = vec![];
    for &element in slice {
        if pred(&element) {
            result.push(element);
        }
    }
    result
}
```

Esto especifica que la referencia en i32 en el límite del rasgo `Fn` puede tener cualquier duración.

Lo siguiente no funciona:

```
fn wrong_copy_if<'a, F>(slice: &[i32], pred: F) -> Vec<i32>
    where F: Fn(&'a i32) -> bool
```

```
{                                   // <-----------------+
    let mut result = vec![];        //       'a scope   |
    for &element in slice {         // <--------+       |
        if pred(&element) {         //          |       |
            result.push(element);   // element's|       |
        }                           //   scope  |       |
    }                               // <--------+       |
    result                          //                  |
}                                   // <-----------------+
```

El compilador da el siguiente error:

```
error: `element` does not live long enough
if pred(&element) {         //          |       |
         ^~~~~~~
```

porque la variable local del `element` no vive tanto como `'a` vida útil (como podemos ver en los comentarios del código).

La vida útil no se puede declarar en el nivel de función, porque necesitamos otra vida útil. Es por eso que usamos `for<'a>` : para especificar que la referencia puede ser válida para cualquier tiempo de vida (por lo tanto, se puede usar un tiempo de vida más pequeño).

Los límites del rasgo de rango superior también se pueden usar en estructuras:

```
struct Window<F>
    where for<'a> F: FnOnce(&'a Window<F>)
{
    on_close: F,
}
```

así como en otros artículos.

Los límites de rasgo de rango más alto son los que se usan más comúnmente con los rasgos `Fn*` .

Para estos ejemplos, el elision de por vida funciona bien, así que no tenemos que especificar los tiempos de vida.

Lea Vidas en línea: https://riptutorial.com/es/rust/topic/2074/vidas

Creditos

S. No	Capítulos	Contributors
1	Empezando con Rust	Andy Hayden, ar-ms, Aurora0001, Community, Cormac O'Brien, D. Ataro, David Grayson, Eric Platon, gavinb, IceyEC, John, Jon Gjengset, Kellen, kennytm, Kevin Montrose, Lukabot, mmstick, Neikos, Pavel Strakhov, Shepmaster, Timidger, Tot Zam, Tshepang, Wolf, xfix, Yohaï Berreby
2	Aplicaciones GUI	eddy, vaartis
3	Archivo I / O	antoyo, Kornel
4	Argumentos de línea de comando	Aurora0001
5	Arreglos, vectores y cortes	antoyo, Aurora0001, John, Matthieu M.
6	Asamblea en línea	4444, Aurora0001
7	Auto-desreferenciación	Aurora0001, John, kennytm, Kornel, Timidger, vaartis, Winger Sendon
8	Bucles	Andy Hayden, apopiak, Arrem, Aurora0001, JDemler, John, kennytm, Mario Carneiro, Matt Smith, Matthieu M., mcarton, Sanpi, Shepmaster, Timidger, Winger Sendon, YOU
9	Carga	Arrem, Aurora0001, Bo Lu, Charlie Egan, Cormac O'Brien, David Grayson, Enigma, John, Lukas Kalbertodt
10	Cierres y expresiones lambda.	xea
11	Constantes asociadas	Ameo, Aurora0001, Hauleth
12	Derivado personalizado: "Macros 1.1"	Vi.
13	Documentación	Aurora0001, Cormac O'Brien, Hauleth
14	estafa	torkleyy
15	Estructuras	4444, Jon Gjengset, letmutx

16	Futuros y Async IO	KolesnichenkoDS
17	Generación de números aleatorios	Phil J. Laszkowicz
18	Genéricos	Kornel, xea
19	Globales	Cormac O'Brien, Jean Pierre Dudey, John, kennytm, Leo Tindall , mcarton
20	Guía de estilo de óxido	Aurora0001, Cldfire, James Gilles, tversteeg
21	Instrumentos de cuerda	Arrem, Aurora0001, David Grayson, KokaKiwi, Lukas Kalbertodt , mcarton, rap-2-h, tmr232, Yos Riady
22	Interfaz de función externa (FFI)	Aurora0001, John, Konstantin V. Salikhov
23	Iteradores	Aurora0001, Chris Emerson, Hauleth, John, Lukas Kalbertodt, Matt Smith, Shepmaster
24	La coincidencia de patrones	adelarsq, Andy Hayden, aSpex, Aurora0001, Cormac O'Brien, Lukas Kalbertodt, mcarton, mnoronha, xea
25	Macros	Aurora0001, kennytm, Matt Smith
26	Manejo de errores	Cormac O'Brien, John, kennytm, Winger Sendon, xea
27	Manejo de señales	Aurora0001, Jean Pierre Dudey, mmstick
28	Marco web de hierro	4444, Aurora0001, Phil J. Laszkowicz
29	Módulos	Aurora0001, Cormac O'Brien, Dumindu Madunuwan, John, KokaKiwi, Kornel, Lu.nemec, xetra11
30	Opción	antoyo, Arrem, Aurora0001, fxlae, Kornel, letmutx, mcarton, Shepmaster
31	Operadores y sobrecargas	Aurora0001, John, Matthieu M.
32	Óxido de metal desnudo	John, mmstick, SplittyDev
33	Pánicos y Desenrollamientos	Aurora0001, Leo Tindall, Timidger
34	Paralelismo	Aurora0001, John, Ruud, xea, zrneely
35	Pautas inseguras	Aurora0001, John

36	PhantomData	Neikos
37	Propiedad	Aurora0001, Jon Gjengset
38	Pruebas	Aurora0001, Cormac O'Brien, IceyEC, JDemler, Jean Pierre Dudey, kennytm, Lu.nemec, mcarton
39	Punteros en bruto	xea
40	Rasgos	a10y, adelarsq, Arrem, Aurora0001, Cormac O'Brien, Hauleth, John, kennytm, Leo Tindall, Matt Smith, Matthieu M., Mylainos, RamenChef, SplittyDev, tversteeg, xea
41	Rasgos de conversión	Cormac O'Brien, Matthieu M.
42	Redes TCP	E_net4
43	Regex	Aurora0001, vaartis
44	Rust orientado a objetos	adelarsq, Aurora0001, Leo Tindall, Marco Alka, Matthieu M., s3rvac, Sorona, Timidger
45	Serde	Aurora0001, dtolnay, kennytm
46	The Drop Rasgo - Destructores en Rust	Leo Tindall, Neikos
47	Tipos de datos primitivos	John
48	Tuplas	adelarsq, Ameo, Aurora0001, John, Jon Gjengset, Matthieu M.
49	Valores en caja	BookOwl
50	Vidas	antoyo, Cormac O'Brien, Jean Pierre Dudey, letmutx, xetra11