Introducción

Esta es una guía (no exhaustiva) para desarrolladores de C# y .NET que son completamente nuevos en el lenguaje de programación Rust. Algunos conceptos y construcciones se traducen bastante bien entre C#/.NET y Rust, pero pueden expresarse de manera diferente, mientras que otros son un cambio radical, como la gestión de la memoria. Esta guía ofrece una breve comparación y mapeo de esas construcciones y conceptos con ejemplos concisos.

Los autores[1] originales de esta guía eran desarrolladores C#/.Net que eran completamente nuevos en Rust. Esta guía es la compilación de conocimiento adquirido por los autores escribiendo código Rust durante varios meses. Es la guía que los autores desearían haber tenido cuando comenzaron su viaje en Rust. Dicho esto, los autores te animarían a leer libros y otro material disponible en la web para abrazar Rust y sus convenciones en lugar de intentar aprenderlo exclusivamente a través del prisma de C# y .NET. Mientras tanto, esta guía puede ayudar a responder algunas preguntas rápidamente, como: *¿Rust soporta herencia, concurrencia, programación asíncrona, etc.?*

Suposiciones:

- El lector es un experimentado desarrollador de C#/.NET.
- El lector es completamente nuevo en Rust.

Objetivos:

- Proporcionar una breve comparación y mapeo de varios temas de C#/.NET con sus contrapartes en Rust.
- Proporcionar enlaces a referencias de Rust, libros y artículos para una lectura adicional sobre los temas.

No objetivos:

- Discusión de patrones de diseño y arquitecturas.
- Tutorial sobre el lenguaje Rust.
- Que el lector sea competente en Rust después de leer esta guía.
- Aunque hay ejemplos cortos que contrastan el código de C# y Rust para algunos temas, esta guía no pretende ser un recetario de recetas de codificación en los dos lenguajes.

---

[1] Los autores originales de esta guía fueron (en orden alfabético): Atif Aziz, Bastian Burger, Daniele

Antonio Maggio, Dariusz Parys and Patrick Schuler.

License

Contribución

Estas invitado a contribuir 💔 a esta guía abriendo issues y enviando pull requests.

Aquí algunas ideas 💡para como y donde tu puedes ayudar más con contribuciones.

- Corrige cualquier error ortográfico o gramatical que encuentres mientras lees.
- Corrige inexactitudes técnicas.
- Soluciona errores lógicos o de compilación en ejemplos de código.
- Mejora el inglés o el español, especialmente si es tu lengua materna o tienes un dominio excelente del idioma.
- Amplía una explicación para proporcionar más contexto o mejorar la claridad de algún tema o concepto.
- Mantén la información actualizada con cambios en C#, .NET y Rust. Por ejemplo, si hay un cambio en C# o Rust que acerca más a los dos lenguajes, algunas partes, incluido el código de muestra, pueden necesitar revisión.

Si estás realizando una corrección pequeña o modesta, como corregir un error ortográfico o un error de sintaxis en un ejemplo de código, siéntete libre de enviar una solicitud de extracción directamente. Para cambios que puedan requerir un esfuerzo considerable de tu parte (y de los revisores como resultado), se recomienda encarecidamente que presentes un issue y busques la aprobación de los mantenedores/editores antes de invertir tu tiempo. Esto evitará desilusiones 💔 en caso de que la solicitud de extracción sea rechazada por diversas razones.

Hacer contribuciones rápidas se ha vuelto muy sencillo. Si ves un error en una página y estás en línea, puedes hacer clic en el ícono de edición 📝 en la esquina de la página para modificar el origen en formato Markdown del contenido y enviar un cambio.

Directrices de Contribución

- Apegarse a los objetivos de esta guía establecidos en la introducción; en otras palabras, ¡evitar los no objetivos!
- Preferir mantener el texto breve y utilizar ejemplos de código cortos, concisos y realistas para ilustrar un punto.

- Siempre que sea posible, proporcionar y comparar ejemplos en Rust y C#.
- Siéntete libre de utilizar las últimas características del lenguaje C#/Rust si hace que un ejemplo sea más simple, conciso y similar en ambos idiomas.
- Evita el uso de paquetes de la comunidad en ejemplos de C#. Apegarse a la Biblioteca Estándar de .NET tanto como sea posible. Dado que la Biblioteca Estándar de Rust tiene una API mucho más pequeña, es más aceptable mencionar crates para alguna funcionalidad, en caso de ser necesario para mostrar un ejemplo (como `rand` para generación de números aleatorios), pero asegúrate de que los crates sean maduros, populares y de confianza.
- Haz que el código de ejemplo sea lo más independiente y ejecutable posible (a menos que la idea sea explicar un error en tiempo de compilación o de ejecución).
- Mantén el estilo general de esta guía, que es evitar usar la palabra *tu* como si se estuviera indicando o instruyendo al lector; en su lugar, utiliza la voz en tercera persona. Por ejemplo, en lugar de decir, “Tu representas datos opcionales en Rust con el tipo `Option<T>` ”, escribe en su lugar, “Rust tiene el tipo `Option<T>` que se utiliza para representar datos opcionales”.

Empezando

Rust Playground

La forma más sencilla de comenzar con Rust sin necesidad de ninguna instalación local es utilizar el Playground de Rust. Es una interfaz de desarrollo mínima que se ejecuta en el navegador web y permite escribir y ejecutar código Rust.

Dev Container

El entorno de ejecución de el Playground de Rust tiene algunas limitaciones, como el tiempo de compilación/ejecución, la memoria y la red. Otra opción que no requiere instalar Rust sería utilizar un *dev container*, como el proporcionado en el repositorio https://github.com/microsoft/vscode-remote-try-rust. Al igual que el Playground de Rust, el contenedor de desarrollo se puede ejecutar directamente en un navegador web utilizando GitHub Codespaces o localmente con Visual Studio Code.

Instalación Local

Para realizar una instalación local completa del compilador Rust y sus herramientas de desarrollo, consulta la sección Instalación del capítulo Empezando en el libro El Lenguaje de Programación Rust, o visita la página de instalación en rust-lang.org.

Lenguaje

Esta sección compara las características de los lenguajes C# y Rust.

Tipos Escalares

La siguiente tabla enumera los tipos primitivos en Rust y su equivalente en C# y .NET:

Rust	C#	.NET	Notas
`bool`	`bool`	`Boolean`	
`char`	`char`	`Char`	Mirar la nota 1.
`i8`	`sbyte`	`SByte`	
`i16`	`short`	`Int16`	
`i32`	`int`	`Int32`	
`i64`	`long`	`Int64`	
`i128`		`Int128`	
`isize`	`nint`	`IntPtr`	
`u8`	`byte`	`Byte`	
`u16`	`ushort`	`UInt16`	
`u32`	`uint`	`UInt32`	
`u64`	`ulong`	`UInt64`	
`u128`		`UInt128`	
`usize`	`nuint`	`UIntPtr`	
`f32`	`float`	`Single`	
`f64`	`double`	`Double`	
	`decimal`	`Decimal`	
`()`	`void`	`Void` o `ValueTuple`	Mirar las notas 2 y 3.
	`object`	`Object`	Mirar la nota 3.

Notas:

1. `char` en Rust y `Char` en .NET tienen diferentes definiciones. En Rust, un `char` tiene 4 bytes de ancho y es un Unicode scalar value, pero en .NET, a `Char` tiene 2 bytes de ancho y almacena el carácter usando la codificación UTF-16. Para más información, mirar la documentación de `char` en Rust.

2. Mientras que en Rust, unit `()` (una tupla vacía) es un *valor expresable*, el equivalente más cercano en C# sería `void` para representar la nada. Sin embargo, `void` no es un *valor expresable*, excepto cuando se usan punteros y código no seguro. .NET tiene `ValueTuple`, que es una tupla vacía, pero C# no tiene una sintaxis literal como `()`

para representarlo. `ValueTuple` se puede usar en C#, pero es muy poco común. A diferencia de C#, F# sí tiene un tipo unit similar a Rust.

3. Mientras `void` y `object` no son tipos escalares (aunque tipos escalares como `int` son subclases de `object` en la jerarquía de tipos de .NET), se han incluido en la tabla anterior por conveniencia.

Mira también:

- Primitivos (Rust By Example)

Strings

Existen dos tipos de strings en Rust: `String` and `&str`. El primero es alocado en el monticulo (heap) y el ultimo es un slice de `String` o un `&str`.

Nota: Slice significa rebana, parte, etc. quiere decir que es una porción de un texto.

La comparación de estos a .NET es mostrada en la siguiente tabla:

Rust	.NET	Nota
`&mut str`	`Span<char>`	
`&str`	`ReadOnlySpan<char>`	
`Box<str>`	`String`	mirar Nota 1.
`String`	`String`	
`String` (mutable)	`StringBuilder`	mirar Nota 1.

Hay diferencias en trabajar con strings en Rust y .Net, pero los equivalentes de arriba deberian de ser un buen punto de inicio. Una de las diferencias es que los strings de Rust son codificados en UTF-8, pero los strings de .NET son codificados en UTF-16. Además los strings de .Net son inmutables, pero los strings en Rust pueden ser mutables cuando se los declara como tal. por ejemplo `let s = &mut String::from("hello");`

Hay también diferencias en usar strings debido al concepto del ownership. Para leer más acerca del ownership con el tipo String, mira el libro de Rust.

Notas

1. El tipo `Box<str>` en Rust es equivalente a el tipo `String` en .NET. La diferencia entre los tipos `Box<str>` y `String` en Rust es que el primero almacena el puntero y el tamaño mientras que el segundo almacena puntero, tamaño y capacidad, permitiendo al `String` crecer en tamaño. Este es similar al el tipo `StringBuilder` de .NET cuando el String de Rust es declarado como mutable.

C#:

```
ReadOnlySpan<char> span = "Hello, World!";
string str = "Hello, World!";
StringBuilder sb = new StringBuilder("Hello, World!");
```

Rust:

```
let span: &str = "Hello, World!";
let str = Box::new("Hello World!");
let mut sb = String::from("Hello World!");
```

String Literales

Las literales de cadena en .NET son tipos `String` inmutables y alocados en el heap (montículo). En Rust, son `&'static str`, que es inmutable, tiene un tiempo de vida global y no se asigna en el montículo; están integradas en el binario compilado.

C#

```
string str = "Hello, World!";
```

Rust

```
let str: &'static str = "Hello, World!";
```

En C# los strings literales de verbatim son equivalentes a los string literales sin procesar en Rust.

C#

```
string str = @"Hello, \World/!";
```

Rust

```
let str = r#"Hello, \World/!"#;
```

En C# los string literales UTF-8 en C# son equivalentes a las string literales de bytes en Rust.

C#

```
ReadOnlySpan<byte> str = "hello"u8;
```

Rust

```
let str = b"hello";
```

Interpolación de Strings

C# tiene una característica incorporada de interpolación de cadenas que te permite incrustar expresiones dentro de una cadena literal. El siguiente ejemplo muestra cómo usar la interpolación de cadenas en C#:

```
string name = "John";
int age = 42;
string str = $"Person {{ Name: {name}, Age: {age} }}";
```

Rust no tiene una característica incorporada de interpolación de cadenas. En su lugar, se utiliza la macro `format!` para formatear una cadena. El siguiente ejemplo muestra cómo usar la interpolación de cadenas en Rust:

```
let name = "John";
let age = 42;
let str = format!("Person {{ name: {name}, age: {age} }}");
```

Las clases y structs personalizados también se pueden interpolar en C# debido a que el método `ToString()` está disponible para cada tipo al heredar de `object`.

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString() =>
        $"Person {{ Name: {Name}, Age: {Age} }}";
}

var person = new Person { Name = "John", Age = 42 };
Console.Writeline(person);
```

En Rust, no hay un formato predeterminado implementado o heredado para cada tipo. En su lugar, se debe implementar el trait `std::fmt::Display` para cada tipo que necesite ser convertido a una cadena.

```
use std::fmt::*;

struct Person {
    name: String,
    age: i32,
}

impl Display for Person {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result {
        write!(f, "Person {{ name: {}, age: {} }}", self.name, self.age)
    }
}

let person = Person {
    name: "John".to_owned(),
    age: 42,
};

println!("{person}");
```

Otra opción es utilizar el trait `std::fmt::Debug`. El trait `Debug` está implementado para todos los tipos estándar y se puede usar para imprimir la representación interna de un tipo. El siguiente ejemplo muestra cómo utilizar el atributo `derive` para imprimir la representación interna de una estructura personalizada utilizando la macro `Debug`. Esta declaración se utiliza para implementar automáticamente el trait `Debug` para la estructura `Person`:

```
#[derive(Debug)]
struct Person {
    name: String,
    age: i32,
}

let person = Person {
    name: "John".to_owned(),
    age: 42,
};

println!("{person:?}");
```

> Nota: El uso del especificador de formato `:?` utilizará el trait `Debug` para imprimir la estructura, mientras que omitirlo utilizará el trait `Display`.

Mira también:

- Rust by Example - Debug

Tipos Estructurados

Tipos de objetos y colecciones comúnmente utilizados en .NET y su mapeo a Rust

C#	Rust
`Array`	`Array`
`List`	`Vec`
`Tuple`	`Tuple`
`Dictionary`	`HashMap`

Array

Los arrays fijos son compatibles de la misma manera en Rust que en .NET.

C#:

```
int[] someArray = new int[2] { 1, 2 };
```

Rust:

```
let someArray: [i32; 2] = [1,2];
```

Listas

En Rust, el equivalente de un `List<T>` es un `Vec<T>`. Los arrays pueden convertirse a Vecs y viceversa.

C#:

```
var something = new List<string>
{
    "a",
    "b"
};

something.Add("c");
```

Rust:

```
let mut something = vec![
    "a".to_owned(),
    "b".to_owned()
];

something.push("c".to_owned());
```

Tuplas

C#:

```
var something = (1, 2)
Console.WriteLine($"a = {something.Item1} b = {something.Item2}");
```

Rust:

```
let something = (1, 2);
println!("a = {} b = {}", something.0, something.1);

// soporta deconstrucción
let (a, b) = something;
println!("a = {} b = {}", a, b);
```

NOTA: En Rust, los elementos de las tuplas no pueden tener nombres como en C#. La única forma de acceder a un elemento de la tupla es utilizando el índice del elemento o desestructurando la tupla.

Diccionarios

En Rust el equivalente de un `Dictionary<TKey, TValue>` es un `Hashmap<K, V>`.

C#:

```
var something = new Dictionary<string, string>
{
    { "Foo", "Bar" },
    { "Baz", "Qux" }
};

something.Add("hi", "there");
```

Rust:

```
let mut something = HashMap::from([
    ("Foo".to_owned(), "Bar".to_owned()),
    ("Baz".to_owned(), "Qux".to_owned())
]);

something.insert("hi".to_owned(), "there".to_owned());
```

Mirar también:

- Biblioteca estándar de Rust - Colecciones

Tipos Personalizados

Las siguientes secciones discuten varios temas y constructos relacionados con el desarrollo de tipos personalizados:

- Classes
- Records
- Estructuras
- Interfaces
- Tipos Enumeración
- Miembros

Classes

Rust no tiene clases. Solo tiene estructuras o `struct` .

Records

Rust no tiene ningún estructura para crear records, ya sea como `record struct` o `record class` en C#.

Estructuras (`struct`)

Las estructuras en Rust y C# comparten algunas similitudes:

- Se definen con la palabra clave `struct`, pero en Rust, `struct` simplemente define los datos/campos. Los aspectos de comportamiento en términos de funciones y métodos se definen por separado en un *bloque de implementación* (`impl`).
- Pueden implementar múltiples traits en Rust de la misma manera que pueden implementar múltiples interfaces en C#.
- No pueden ser subclasificadas.
- Se asignan en la pila (stack) por defecto, a menos que:
  - En .NET, se haga boxing o se castee a una interfaz.
  - En Rust, se envuelvan en un puntero inteligente como `Box`, `Rc` / `Arc`.

En C#, un `struct` es una forma de modelar un *value type* (tipos de valor) en .NET, que suele ser algún primitivo específico del dominio o compuesto con semántica de igualdad de valores. En Rust, un `struct` es la construcción principal para modelar cualquier estructura de datos (la otra siendo un `enum`).

Un `struct` (o `record struct`) en C# tiene copia por valor y semántica de igualdad de valores por defecto, pero en Rust, esto requiere simplemente un paso más utilizando el atributo `#derive` y enumerando los traits que se deben implementar:

```
#[derive(Clone, Copy, PartialEq, Eq, Hash)]
struct Point {
    x: i32,
    y: i32,
}
```

En C#/.NET, los Value Types suelen ser diseñados por un desarrollador para ser inmutables. Se considera una práctica recomendada desde el punto de vista semántico, pero el lenguaje no impide diseñar un `struct` que realice modificaciones destructivas o en el lugar. En Rust, es lo mismo. Un tipo debe ser conscientemente desarrollado para ser inmutable.

Dado que Rust no tiene clases y, en consecuencia, jerarquías de tipos basadas en la subclase, el comportamiento compartido se logra mediante traits y genéricos, y el polimorfismo a través de la despacho virtual utilizando trait objects.

Considera la siguiente `struct` que representa un rectángulo en C#:

```
struct Rectangle
{
    public Rectangle(int x1, int y1, int x2, int y2) =>
        (X1, Y1, X2, Y2) = (x1, y1, x2, y2);

    public int X1 { get; }
    public int Y1 { get; }
    public int X2 { get; }
    public int Y2 { get; }

    public int Length => Y2 - Y1;
    public int Width => X2 - X1;

    public (int, int) TopLeft => (X1, Y1);
    public (int, int) BottomRight => (X2, Y2);

    public int Area => Length * Width;
    public bool IsSquare => Width == Length;

    public override string ToString() => $"({X1}, {Y1}), ({X2}, {Y2})";
}
```

El equivalente en Rust sería:

```rust
#![allow(dead_code)]

struct Rectangle {
    x1: i32, y1: i32,
    x2: i32, y2: i32,
}

impl Rectangle {
    pub fn new(x1: i32, y1: i32, x2: i32, y2: i32) -> Self {
        Self { x1, y1, x2, y2 }
    }

    pub fn x1(&self) -> i32 { self.x1 }
    pub fn y1(&self) -> i32 { self.y1 }
    pub fn x2(&self) -> i32 { self.x2 }
    pub fn y2(&self) -> i32 { self.y2 }

    pub fn length(&self) -> i32 {
        self.y2 - self.y1
    }

    pub fn width(&self)  -> i32 {
        self.x2 - self.x1
    }

    pub fn top_left(&self) -> (i32, i32) {
        (self.x1, self.y1)
    }

    pub fn bottom_right(&self) -> (i32, i32) {
        (self.x2, self.y2)
    }

    pub fn area(&self)  -> i32 {
        self.length() * self.width()
    }

    pub fn is_square(&self)  -> bool {
        self.width() == self.length()
    }
}

use std::fmt::*;

impl Display for Rectangle {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result {
        write!(f, "({}, {}), ({}, {})", self.x1, self.y2, self.x2, self.y2)
    }
}
```

Ten en cuenta que un `struct` en C# hereda el método `ToString` de `object` y, por lo tanto,

anula la implementación base para proporcionar una representación de cadena personalizada. Dado que no hay herencia en Rust, la forma en que un tipo indica el soporte para alguna representación *formateada* es mediante la implementación del trait `Display`. Esto permite que una instancia de la estructura participe en el formateo, como se muestra en la llamada a `println!` a continuación:

```
fn main() {
    let rect = Rectangle::new(12, 34, 56, 78);
    println!("Rectangle = {rect}");
}
```

Interfaces

Rust no tiene interfaces como las que se encuentran en C#/.NET. En su lugar, tiene *traits*. Similar a una interfaz, un trait representa una abstracción y sus miembros forman un contrato que debe cumplirse cuando se implementa en un tipo.

Al igual que las interfaces pueden tener métodos predeterminados en C#/.NET (donde se proporciona un cuerpo de implementación predeterminado como parte de la definición de la interfaz), los traits en Rust también pueden tenerlos. El tipo que implementa la interfaz/trait puede proporcionar posteriormente una implementación más adecuada y/o optimizada.

Las interfaces en C#/.NET pueden tener todo tipo de miembros, desde propiedades, indexadores y eventos hasta métodos, tanto estáticos como de instancia. De manera similar, los traits en Rust pueden tener métodos (de instancia), funciones asociadas (piensa en métodos estáticos en C#/.NET) y constantes.

Además de las jerarquías de clases, las interfaces son un medio fundamental para lograr el polimorfismo mediante la despacho dinámico para abstracciones transversales. Permiten escribir código de propósito general contra las abstracciones representadas por las interfaces sin tener en cuenta mucho los tipos concretos que las implementan. Lo mismo se puede lograr con los *Trait Objects* en Rust de manera limitada. Un trait object es esencialmente una *v-table* identificada con la palabra clave `dyn` seguida del nombre del trait, como en `dyn Shape` (donde `Shape` es el nombre del trait). Los trait objects siempre viven detrás de un puntero, ya sea una referencia (por ejemplo, `&dyn Shape`) o el `Box` asignado en el montón (por ejemplo, `Box<dyn Shape>`). Esto es algo similar a en .NET, donde una interfaz es un tipo de referencia, de modo que un tipo de valor convertido a una interfaz se coloca automáticamente en la montón gestionado. La limitación de paso de los trait objects mencionada anteriormente es que el tipo de implementación original no se puede recuperar. En otras palabras, mientras que es bastante común hacer un *downcast* o probar si una interfaz es una instancia de alguna otra interfaz o tipo subyacente o concreto, lo mismo no es posible en Rust (sin esfuerzo y soporte adicionales).

> Cuando hablamos de downcasting nos referimos al poder obtener a base de una abstracción un tipo concreto.

Puedes mirar también:

- Traits: Definiendo Comportamiento Compartido - El Lenguaje de Programación Rust

- Downcast Trait Object
- Downcasting in Rust

Tipos Enumeración (`enum`)

En C#, un `enum` es un tipo de valor que asigna nombres simbólicos a valores enteros:

```
enum DayOfWeek
{
    Sunday = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
}
```

Rust tiene una sintaxis prácticamente *idéntica* para hacer lo mismo:

```
enum DayOfWeek
{
    Sunday = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
}
```

A diferencia de en .NET, una instancia de un tipo `enum` en Rust no tiene ningún comportamiento predefinido que se herede. Ni siquiera puede participar en comprobaciones de igualdad tan simples como `dow == DayOfWeek::Friday`. Para hacerlo en cierta medida comparable en función con un `enum` en C#, utiliza el atributo `#derive` para que los macros implementen automáticamente la funcionalidad comúnmente necesaria:

```
#[derive(Debug,     // habilita el formateo en "{:?}"
         Clone,     // requerido por Copy
         Copy,      // habilita la semántica de copia por valor
         Hash,      // habilita la posibilidad de usar en tipos de mapa
         PartialEq  // habilita la igualdad de valores (==)
)]
enum DayOfWeek
{
    Sunday = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
}

fn main() {
    let dow = DayOfWeek::Wednesday;
    println!("Day of week = {dow:?}");

    if dow == DayOfWeek::Friday {
        println!("Yay! It's the weekend!");
    }

    // coerce to integer
    let dow = dow as i32;
    println!("Day of week = {dow:?}");

    let dow = dow as DayOfWeek;
    println!("Day of week = {dow:?}");
}
```

Como muestra el ejemplo anterior, un `enum` puede ser convertido a su valor integral asignado, pero lo contrario no es posible como en C# (aunque esto a veces tiene la desventaja en C#/.NET de que una instancia de `enum` puede contener un valor no representado). En su lugar, depende del desarrollador proporcionar una función auxiliar de este tipo:

```
impl DayOfWeek {
    fn from_i32(n: i32) -> Result<DayOfWeek, i32> {
        use DayOfWeek::*;
        match n {
            0 => Ok(Sunday),
            1 => Ok(Monday),
            2 => Ok(Tuesday),
            3 => Ok(Wednesday),
            4 => Ok(Thursday),
            5 => Ok(Friday),
            6 => Ok(Saturday),
            _ => Err(n)
        }
    }
}
```

La función `from_i32` devuelve un `DayOfWeek` en un `Result` indicando éxito (`Ok`) si `n` es válido. De lo contrario, devuelve `n` tal cual en un `Result` que indica fallo (`Err`):

```
let dow = DayOfWeek::from_i32(5);
println!("{dow:?}"); // prints: Ok(Friday)

let dow = DayOfWeek::from_i32(50);
println!("{dow:?}"); // prints: Err(50)
```

Existen crates en Rust que pueden ayudar a implementar este mapeo a partir de tipos integrales en lugar de tener que codificarlos manualmente.

Un tipo `enum` en Rust también puede servir como una forma de diseñar tipos de unión (discriminados), que permiten que diferentes *variantes* contengan datos específicos para cada variante. Por ejemplo:

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

Esta forma de declaración de `enum` no existe en C#, pero se puede emular con registros (class records):

```
var home = new IpAddr.V4(127, 0, 0, 1);
var loopback = new IpAddr.V6("::1");

abstract record IpAddr
{
    public sealed record V4(byte A, byte B, byte C, byte D): IpAddr;
    public sealed record V6(string Address): IpAddr;
}
```

La diferencia entre ambas es que la definición en Rust produce un *tipo cerrado* sobre las variantes. En otras palabras, el compilador sabe que no habrá otras variantes de `IpAddr` excepto `IpAddr::V4` y `IpAddr::V6`, y puede utilizar ese conocimiento para realizar verificaciones más estrictas. Por ejemplo, en una expresión `match` que es similar a la expresión `switch` en C#, el compilador de Rust generará un error a menos que se cubran todas las variantes. En cambio, la emulación con C# crea realmente una jerarquía de clases (aunque expresada de manera muy concisa) y, dado que `IpAddr` es una *clase base abstracta*, el conjunto de todos los tipos que puede representar es desconocido para el compilador.

Miembros

Constructores

Rust no tiene ninguna noción de constructores. En su lugar, simplemente escribes funciones factory que retornan una instancia del tipo. Las funciones Factory pueden ser independientes o *funciones asociadas* al tipo. En términos de C# las funciones asociadas son como tener metodos estaticos en un tipo. Por convención, si hay solo una función factory para una estructura, se le llama `new`:

```
struct Rectangle {
    x1: i32, y1: i32,
    x2: i32, y2: i32,
}

impl Rectangle {
    pub fn new(x1: i32, y1: i32, x2: i32, y2: i32) -> Self {
        Self { x1, y1, x2, y2 }
    }
}
```

Dado que las funciones en Rust (ya sean asociadas u otras) no admiten sobrecarga; las funciones factory tienen que tener nombres únicos. Por ejemplo, a continuación se presentan algunos ejemplos de las funciones constructores o factory disponibles en `String`.

- `String::new`: crea un string vació.
- `String::with_capacity`: crea un string con una capacidad de buffer inicial.
- `String::from_utf8`: crea un string desde bytes de texto codificado en UTF-8.
- `String::from_utf16`: crea un string desde bytes de texto codificado en UTF-16.

En el caso de un tipo `enum` en Rust, las variantes actúan como constructores. Mira [la sección de tipos Enumerados][ennums] para ver más.

Mira también:

- Constructors are static, inherent methods (C-CTOR)

Métodos (estáticos y basados en instancias)

Al igual que en C#, los tipos de Rust (tanto `enum` como `struct`) pueden tener métodos estáticos y basados en instancias. En la terminología de Rust, un método siempre es basado en instancia y se identifica por el hecho de que su primer parametro se llama `self`. El parametro `self` no tiene una anotación de tipo, ya que siempre es el tipo al que pertenece el método. Un método estático se llama función asociada. En el ejemplo de a continuación, `new` es una función asociada y el resto (`length`, `width`, y `area`) son métodos de el tipo.

```
struct Rectangle {
    x1: i32, y1: i32,
    x2: i32, y2: i32,
}

impl Rectangle {
    pub fn new(x1: i32, y1: i32, x2: i32, y2: i32) -> Self {
        Self { x1, y1, x2, y2 }
    }

    pub fn length(&self) -> i32 {
        self.y2 - self.y1
    }

    pub fn width(&self)  -> i32 {
        self.x2 - self.x1
    }

    pub fn area(&self)  -> i32 {
        self.length() * self.width()
    }
}
```

Constantes

Al igual que en C#, un tipo en Rust puede tener constantes. Sin embargo, el aspecto más interesante de notar es que Rust permite que una instancia de tipo se defina como una constante.

```
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    const ZERO: Point = Point { x: 0, y: 0 };
}
```

En C#, lo mismo requeriría un campo de solo lectura estático.

```
readonly record struct Point(int X, int Y)
{
    public static readonly Point Zero = new(0, 0);
}
```

Eventos

Rust no tiene soporte incorporado para que los miembros de tipo anuncien y disparen eventos, como lo tiene C# con la palabra clave `event`.

Propiedades

En C#, los campos de un tipo suelen ser privados. Luego, se protegen/encapsulan mediante miembros de propiedades miembro como métodos de acceso (get y set) para leer o escribir, para validar el valor al establecerlo o calcular un valor al leerlo. Rust solo tiene métodos donde un getter tiene el mismo nombre que el campo (En Rust, los nombres de los métodos pueden compartir el mismo identificador que un campo) y el setter utiliza un prefijo `set_`

A continuación, se muestra un ejemplo que muestra cómo suelen lucir los métodos de acceso similares a propiedades para un tipo en Rust:

```
struct Rectangle {
    x1: i32, y1: i32,
    x2: i32, y2: i32,
}

impl Rectangle {
    pub fn new(x1: i32, y1: i32, x2: i32, y2: i32) -> Self {
        Self { x1, y1, x2, y2 }
    }

    // como getters de propiedades (cada uno comparte el mismo nombre que el
campo)

    pub fn x1(&self) -> i32 { self.x1 }
    pub fn y1(&self) -> i32 { self.y1 }
    pub fn x2(&self) -> i32 { self.x2 }
    pub fn y2(&self) -> i32 { self.y2 }

    // como setters de propiedades

    pub fn set_x1(&mut self, val: i32) { self.x1 = val }
    pub fn set_y1(&mut self, val: i32) { self.y1 = val }
    pub fn set_x2(&mut self, val: i32) { self.x2 = val }
    pub fn set_y2(&mut self, val: i32) { self.y2 = val }

    // como propiedades calculadas

    pub fn length(&self) -> i32 {
        self.y2 - self.y1
    }

    pub fn width(&self)  -> i32 {
        self.x2 - self.x1
    }

    pub fn area(&self)  -> i32 {
        self.length() * self.width()
    }
}
```

Métodos de Extensión

Los métodos de extensión en C# permiten al desarrollador adjuntar nuevos métodos vinculados estáticamente a tipos existentes, sin necesidad de modificar la definición original del tipo. En el siguiente ejemplo de C#, se añade un nuevo método `Wrap` a la clase `StringBuilder` *mediante una extensión*:

```
using System;
using System.Text;
using Extensions; // (1)

var sb = new StringBuilder("Hello, World!");
sb.Wrap(">>> ", " <<<"); // (2)
Console.WriteLine(sb.ToString()); // Muestra: >>> Hello, World! <<<

namespace Extensions
{
    static class StringBuilderExtensions
    {
        public static void Wrap(this StringBuilder sb,
                                string left, string right) =>
            sb.Insert(0, left).Append(right);
    }
}
```

Ten en cuenta que para que un método de extensión esté disponible (2), se debe importar el namespace con el tipo que contiene el método de extensión (1). Rust ofrece una facilidad muy similar a través de traits, llamada *extension traits*. El siguiente ejemplo en Rust es equivalente al ejemplo de C# anterior; extiende `String` con el método `wrap`:

```
#![allow(dead_code)]

mod exts {
    pub trait StrWrapExt {
        fn wrap(&mut self, left: &str, right: &str);
    }

    impl StrWrapExt for String {
        fn wrap(&mut self, left: &str, right: &str) {
            self.insert_str(0, left);
            self.push_str(right);
        }
    }
}

fn main() {
    use exts::StrWrapExt as _; // (1)

    let mut s = String::from("Hello, World!");
    s.wrap(">>> ", " <<<"); // (2)
    println!("{s}"); // Prints: >>> Hello, World! <<<
}
```

Al igual que en C#, para que el método en el trait de extensión esté disponible (2), el trait de extensión debe importarse (1). También ten en cuenta que el identificador del trait de extensión `StrWrapExt` puede descartarse mediante `_` en el momento de la importación sin

afectar la disponibilidad de `wrap` para `String`.

Modificadores de Visibilidad/Acceso

C# tiene varios modificadores de accesibilidad o visibilidad:

- `private`
- `protected`
- `internal`
- `protected internal` (familia)
- `public`

En Rust, una compilación se compone de un árbol de módulos en el que los módulos contienen y definen *elementos* como tipos, traits, enums, constantes y funciones. Casi todo es privado por defecto. Una excepción es, por ejemplo, *elementos asociados* en un trait público, que son públicos por defecto. Esto es similar a cómo los miembros de una interfaz de C# declarados sin ningún modificador público en el código fuente son públicos por defecto. Rust solo tiene el modificador `pub` para cambiar la visibilidad con respecto al árbol de módulos. Hay variaciones de `pub` que cambian el alcance de la visibilidad pública:

- `pub(self)`
- `pub(super)`
- `pub(crate)`
- `pub(in PATH)`

Para obtener más detalles, consulta la sección Visibility and Privacy en la referencia de Rust.

La tabla a continuación es una aproximación de la correspondencia entre los modificadores de C# y Rust:

C#	Rust	Note
`private`	(default)	Mirar nota 1.
`protected`	N/A	Mirar nota 2.
`internal`	`pub(crate)`	
`protected internal` (familia)	N/A	Mirar nota 2.
`public`	`pub`	

1. No existe una palabra clave para denotar visibilidad privada; es la configuración predeterminada en Rust.

2. Dado que no hay jerarquías de tipos basadas en clases en Rust, no hay un equivalente de `protected`.

Mutabilidad

Al diseñar un tipo en C#, es responsabilidad del desarrollador decidir si un tipo es mutable o inmutable; si admite mutaciones destructivas o no destructivas. C# admite un diseño inmutable para tipos con una *positional record declaration* (`record class` o `readonly record struct`). En Rust, la mutabilidad se expresa en los métodos a través del tipo del parámetro `self`, como se muestra en el siguiente ejemplo:

```
struct Point { x: i32, y: i32 }

impl Point {
    pub fn new(x: i32, y: i32) -> Self {
        Self { x, y }
    }

    // self no es mutable

    pub fn x(&self) -> i32 { self.x }
    pub fn y(&self) -> i32 { self.y }

    // self es mutable

    pub fn set_x(&mut self, val: i32) { self.x = val }
    pub fn set_y(&mut self, val: i32) { self.y = val }
}
```

En C#, puedes realizar mutaciones no destructivas usando `with`:

```
var pt = new Point(123, 456);
pt = pt with { X = 789 };
Console.WriteLine(pt.ToString()); // Muestra: Point { X = 789, Y = 456 }

readonly record struct Point(int X, int Y);
```

No hay `with` en Rust, pero para emular algo similar en Rust, debe estar integrado en el diseño del tipo:

```rust
struct Point { x: i32, y: i32 }

impl Point {
    pub fn new(x: i32, y: i32) -> Self {
        Self { x, y }
    }

    pub fn x(&self) -> i32 { self.x }
    pub fn y(&self) -> i32 { self.y }

    // los siguientes métodos consumen self y devuelven una nueva instancia:

    pub fn set_x(self, val: i32) -> Self { Self::new(val, self.y) }
    pub fn set_y(self, val: i32) -> Self { Self::new(self.x, val) }
}
```

Funciones Locales

C# y Rust ofrecen funciones locales, pero las funciones locales en Rust están limitadas al equivalente de funciones locales estáticas en C#. En otras palabras, las funciones locales en Rust no pueden usar variables de su ámbito léxico circundante; pero las *closures* pueden.

También puedes ver:

- Closures: Funciones anónimas que capturan su entorno - El Lenguaje de Programación Rust

Lambda y Closures

C# y Rust permiten que las funciones sean utilizadas como valores de primera clase que posibilitan la escritura de *funciones de orden superior*. Las funciones de orden superior son esencialmente funciones que aceptan otras funciones como argumentos para permitir que el llamador participe en el código de la función llamada. En C#, los *function pointers seguros* se representan mediante delegados, siendo los más comunes `Func` y `Action`. El lenguaje C# permite la creación de instancias ad hoc de estos delegados a través de *expresiones lambda*.

Rust también tiene function pointers, siendo el tipo más simple `fn`:

```
fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(|x| x + 1, 5);
    println!("The answer is: {}", answer); // Prints: The answer is: 12
}
```

Sin embargo, Rust hace una distinción entre *funciones punteros* (donde `fn` define un tipo) y *closures*: una closure puede hacer referencia a variables desde su ámbito léxico circundante, pero no una función puntero. Aunque C# también tiene function pointers (`*delegate`), el equivalente gestionado y seguro para el tipo sería una expresión lambda estática.

Las funciones y métodos que aceptan closures se escriben con tipos genéricos que está vinculado a uno de los traits que representa funciones: `Fn`, `FnMut` y `FnOnce`. Cuando es el momento de proporcionar un valor para un puntero a función o un cierre, un desarrollador de Rust utiliza una *closure expression* (como `|x| x + 1` en el ejemplo anterior), que se traduce de la misma manera que una expresión lambda en C#. Si la closure expression crea una pointer function
o una closure depende de si la closure expression hace referencia a su contexto o no.

Cuando una closure captura variables de su entorno, entran en juego las reglas de ownership porque el ownership termina en la closure. Para obtener más información, consulta la sección "Moviendo valores capturados fuera de los closures y los traits Fn" de "El Lenguaje de Programación Rust".

Variables

Considera el siguiente ejemplo acerca de asignación de variables en C#:

```
int x = 5;
```

Y el mismo en Rust:

```
let x: i32 = 5;
```

Hasta este momento, la única diferencia visible entre los dos lenguajes es que la posición de la declaración del tipo es diferente. También, ambos, C# y Rust son type-safe: el compilador garantiza que los valores almacenados en una variable tiene siempre el mismo tipo designado. El ejemplo puede ser simplificado por usar la habilidad del compilador para automáticamente inferir el tipo de la variable. En C#:

```
var x = 5;
```

En Rust:

```
let x = 5;
```

Cuando ampliamos el primer ejemplo para cambiar el valor de la variable (reasignamiento), el comportamiento de C# y Rust difieren:

```
var x = 5;
x = 6;
Console.WriteLine(x); // 6
```

En Rust, la misma sentencia no compila:

```
let x = 5;
x = 6; // Error: cannot assign twice to immutable variable 'x'.
println!("{}", x);
```

En Rust, las variables son *inmutables* por defecto. Una vez que un valor es vinculado a un nombre, la variable no puede ser cambiada. Las variables pueden ser *mutables* por adición de `mut` al comienzo de la declaración.

```rust
let mut x = 5;
x = 6;
println!("{}", x); // 6
```

Rust ofrece una alternativa para arreglar este ejemplo de encima que no requiere mutabilidad mediante *shadowing*:

```rust
let x = 5;
let x = 6;
println!("{}", x); // 6
```

C# también soporta shadowing, por ejemplo variables locales pueden ocultar campos y variables miembros del tipo base. En Rust, el ejemplo de arriba demuestra que el shadowing también permite cambiar el tipo sin cambiar el nombre, esto puede ser util si solo queremos transformar el dato en uno con diferente tipo y forma sin tener que tener una variable con distinto nombre en cada ocasión.

Puedes ver también:

- Data races y race conditions para más información acerca de las implicaciones de la mutabilidad
- Scope y shadowing
- Memory management para explicaciones acerca de *moving* y *ownership*

Namespaces

En .NET, se utilizan namespaces para organizar tipos, así como para controlar el ámbito de los tipos y métodos en proyectos.

En Rust, el término "namespace" se refiere a un concepto diferente. El equivalente de un namespace en Rust es un módulo. Tanto en C# como en Rust, la visibilidad de los elementos se puede restringir mediante modificadores de acceso o modificadores de visibilidad, respectivamente. En Rust, la visibilidad predeterminada es *privada* (con solo algunas excepciones). El equivalente al `public` de C# es `pub` en Rust, y `internal` se corresponde con `pub(crate)`. Para un control de acceso más detallado, consulta la referencia de modificadores de visibilidad.

Equivalencia

Cuando se compara por igualdad en C#, esto se refiere a probar la *equivalencia* en algunos casos (también conocida como *igualdad de valor*), y en otros casos se refiere a probar la *igualdad de referencia*, que verifica si dos variables se refieren al mismo objeto subyacente en memoria. Cada tipo personalizado puede ser comparado por igualdad porque hereda de `System.Object` (o `System.ValueType` para tipos de valor, que a su vez hereda de `System.Object`), utilizando cualquiera de las semánticas mencionadas anteriormente.

Por ejemplo, al comparar por equivalencia e igualdad de referencia en C#:

```
var a = new Point(1, 2);
var b = new Point(1, 2);
var c = a;
Console.WriteLine(a == b); // (1) True
Console.WriteLine(a.Equals(b)); // (1) True
Console.WriteLine(a.Equals(new Point(2, 2))); // (1) True
Console.WriteLine(ReferenceEquals(a, b)); // (2) False
Console.WriteLine(ReferenceEquals(a, c)); // (2) True

record Point(int X, int Y);
```

1. El operador de igualdad `==` y el método `Equals` en el `record Point` comparan por igualdad de valor, ya que los registros admiten la igualdad de tipo valor de forma predeterminada.

2. Comparar por igualdad de referencia verifica si las variables se refieren al mismo objeto subyacente en memoria.

Equivalente en Rust:

```
#[derive(Copy, Clone)]
struct Point(i32, i32);

fn main() {
    let a = Point(1, 2);
    let b = Point(1, 2);
    let c = a;
    println!("{}", a == b); // Error: "an implementation of `PartialEq<_>`
might be missing for `Point`"
    println!("{}", a.eq(&b));
    println!("{}", a.eq(&Point(2, 2)));
}
```

El error del compilador anterior ilustra que en Rust las comparaciones de igualdad *siempre*

están relacionadas con una implementación de trait. Para admitir una comparación usando `==`, un tipo debe implementar `PartialEq`.

Corregir el ejemplo anterior significa derivar `PartialEq` para `Point`. Por defecto, al derivar `PartialEq` se compararán todos los campos para la igualdad, por lo que ellos mismos deben implementar `PartialEq`. Esto es comparable a la igualdad de registros en C#.

```
#[derive(Copy, Clone, PartialEq)]
struct Point(i32, i32);

fn main() {
    let a = Point(1, 2);
    let b = Point(1, 2);
    let c = a;
    println!("{}", a == b); // true
    println!("{}", a.eq(&b)); // true
    println!("{}", a.eq(&Point(2, 2))); // false
    println!("{}", a.eq(&c)); // true
}
```

Véase también:

- `Eq` para una versión más estricta de `PartialEq`

Genéricos

Los genéricos en C# proporcionan una forma de crear definiciones para tipos y métodos que pueden ser parametrizados sobre otros tipos. Esto mejora la reutilización de código, la seguridad de tipos y el rendimiento (por ejemplo, evita conversiones en tiempo de ejecución). Considera el siguiente ejemplo de un tipo genérico que agrega una marca de tiempo a cualquier valor:

```
using System;

sealed record Timestamped<T>(DateTime Timestamp, T Value)
{
    public Timestamped(T value) : this(DateTime.UtcNow, value) { }
}
```

Rust también tiene genéricos, como se muestra en el equivalente del ejemplo anterior:

```
use std::time::*;

struct Timestamped<T> { value: T, timestamp: SystemTime }

impl<T> Timestamped<T> {
    fn new(value: T) -> Self {
        Self { value, timestamp: SystemTime::now() }
    }
}
```

Mira también:

- Tipos de Datos Genéricos

Restricciones de tipo genérico

En C#, los tipos genéricos pueden ser restringidos usando la palabra clave where. El siguiente ejemplo muestra tales restricciones en C#:

```
using System;

// Nota: los registros implementan automáticamente `IEquatable`. La siguiente
// implementación muestra esto explícitamente para una comparación con Rust.
sealed record Timestamped<T>(DateTime Timestamp, T Value) :
    IEquatable<Timestamped<T>>
    where T : IEquatable<T>
{
    public Timestamped(T value) : this(DateTime.UtcNow, value) { }

    public bool Equals(Timestamped<T>? other) =>
        other is { } someOther
        && Timestamp == someOther.Timestamp
        && Value.Equals(someOther.Value);

    public override int GetHashCode() => HashCode.Combine(Timestamp, Value);
}
```

Lo mismo se puede lograr en Rust:

```
use std::time::*;

struct Timestamped<T> { value: T, timestamp: SystemTime }

impl<T> Timestamped<T> {
    fn new(value: T) -> Self {
        Self { value, timestamp: SystemTime::now() }
    }
}

impl<T> PartialEq for Timestamped<T>
    where T: PartialEq {
    fn eq(&self, other: &Self) -> bool {
        self.value == other.value && self.timestamp == other.timestamp
    }
}
```

En Rust, las restricciones de tipo genérico se llaman bounds.

En la versión de C#, las instancias de `Timestamped<T>` solo pueden crearse para `T` que implementen `IEquatable<T>` ellos mismos, pero ten en cuenta que la versión de Rust es más flexible porque `Timestamped<T>` *implementa condicionalmente* `PartialEq`. Esto significa que las instancias de `Timestamped<T>` aún pueden crearse para algunos `T` que no son equiparables, pero entonces `Timestamped<T>` no implementará la igualdad a través de `PartialEq` para dicho `T`.

Mira también:

Polimorfismo

Rust no admite clases y herencia, por lo tanto, el polimorfismo no se puede lograr de manera idéntica a C#.

Consulta también:

- Despacho virtual utilizando *objetos de trait*, como se explica en la sección Estructuras
- Genéricos
- Herencia
- Sobrecarga de operadores

Herencia

Como se explica en la sección Estructuras, Rust no proporciona herencia (basada en clases) como en C#. Una forma de proporcionar un comportamiento compartido entre structs es mediante el uso de traits. Sin embargo, similar a la *herencia de interfaces* en C#, Rust permite definir relaciones entre traits mediante el uso de *supertraits*.

Manejo de Excepciones

En .Net, una excepción es un tipo que hereda de la clase `System.Exception`. Excepción es lanzada si un problema ocurre en una sección de código. Un lanzamiento de excepción es pasado hacia arriba al stack hasta que la aplicación la maneje o el programa termine.

Rust no tiene excepciones, pero distingue entre errores *recuperables* y *no recuperables* en su lugar. Un error recuperable representa un problema que debe ser reportado, pero sin embargo el programa continua. Resultado de operaciones que pueden fallar con errores recuperables son de tipo `Result<T, E>`, en donde `E` es del tipo de variante de error. La macro `panic!` detiene la ejecución cuando el programa encuentra un error irrecuperable. Un error irrecuperable es siempre un síntoma de un bug.

Tipos de errores personalizados

En .Net, una excepción personalizada deriva de la clase `Exception`. La documentación en Cómo crear excepciones definidas por el usuario menciona el siguiente ejemplo:

```
public class EmployeeListNotFoundException : Exception
{
    public EmployeeListNotFoundException() { }

    public EmployeeListNotFoundException(string message)
        : base(message) { }

    public EmployeeListNotFoundException(string message, Exception inner)
        : base(message, inner) { }
}
```

En Rust, uno puede implementar el comportamiento básico para los valores erróneos via implementación de el trait `Error`. La implementación minima definida por el usuario en Rust:

```rust
#[derive(Debug)]
pub struct EmployeeListNotFound;

impl std::fmt::Display for EmployeeListNotFound {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        f.write_str("No se pudo encontrar empleado en la lista.")
    }
}

impl std::error::Error for EmployeeListNotFound {}
```

El equivalente para la `Exception.InnerException` de .Net es el método `Error::source()` en Rust. Sin embargo, este no requiere proveer una implementación para `Error::source()`, la implementación por defecto (blanket implementation) retorna un `None`.

Elevando excepciones

Para elevar una excepción en C#, lanza una instancia de la excepción:

```csharp
void ThrowIfNegative(int value)
{
    if (value < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(value));
    }
}
```

Para recuperar errores en Rust, retorna una variante de `Ok` o de `Err` desde un método:

```rust
fn error_if_negative(value: i32) -> Result<(), &'static str> {
    if value < 0 {
        Err("El argumento especificado esta fuera del rango de valores validos.
(Parámetro 'value')")
    } else {
        Ok(())
    }
}
```

La macro `panic!` crea errores irrecuperables:

```rust
fn panic_if_negative(value: i32) {
    if value < 0 {
        panic!("El argumento especificado esta fuera del rango de valores
validos. (Parámetro 'value')")
    }
}
```

Propagación de error

En .Net, excepciones son pasadas hacia arriba hasta que son tratadas o el programa termina. En Rust, los errores irrecuperables son similares, pero tratarlos es poco común.

Los errores recuperables, sin embargo necesitan ser propagados y tratarlos explícitamente. Están presentes siempre representados en la firma de funciones o métodos en Rust. Capturar las excepciones te permiten tomar acciones basadas en la presencia o ausencia de errores en C#.

```csharp
void Write()
{
    try
    {
        File.WriteAllText("file.txt", "content");
    }
    catch (IOException)
    {
        Console.WriteLine("Escribiendo el archivo fallo.");
    }
}
```

En Rust, esto es un equivalente aproximado:

```rust
fn write() {
    match std::fs::File::create("temp.txt")
        .and_then(|mut file| std::io::Write::write_all(&mut file, b"content"))
    {
        Ok(_) => {}
        Err(_) => println!("Escribiendo el archivo fallo."),
    };
}
```

Frecuentemente, los errores recuperables necesitan ser propagados en lugar de ser tratados. Para esto, la firma del metodo necesita ser compatible con el tipo de error propagado. El operador `?` propaga errores ergonómicamente:

```rust
fn write() -> Result<(), std::io::Error> {
    let mut file = std::fs::File::create("file.txt")?;
    std::io::Write::write_all(&mut file, b"content")?;
    Ok(())
}
```

Nota: Para propagar un error con el *question mark operator* la implementación del error necesita ser *compatible*, como describimos en *un atajo para la propagación de errores*. El tipo más "compatible" es el trait object error `Box<dyn Error>`.

Stack traces

Lanzar una excepción no capturada en .Net causara que el runtime imprima un stack trace que permitirá depurar el problema con contexto adicional.

Para errores irrecuperables en Rust, los backtraces de `panic!` ofrecen un comportamiento similar.

Los errores recuperables en Rust estable no soportan aún los backtraces, pero es soportado en Rust experimental cuando usamos el método provide.

Nulabilidad y Opcionalidad

En C#, `null` es a veces usado para representar un valor que es faltante, ausente o lógicamente no inicializado. Por ejemplo:

```
int? some = 1;
int? none = null;
```

Rust no tiene `null` y consecuencialmente contexto no nulleable para habilitar. Los valores opcionales o faltantes son representados por `Option<T>` en su lugar. El equivalente de el código C# de arriba en Rust debería ser:

```
let some: Option<i32> = Some(1);
let none: Option<i32> = None;
```

`Option<T>` en Rust es prácticamente idéntico a `'T option` de F#.

Flujo de Control con Opcionabilidad

En C#, tal vez estes usando sentencias `if` / `else` para controlar el flujo cuando uses valores nulleables.

```
uint? max = 10;
if (max is { } someMax)
{
    Console.WriteLine($"El máximo es {someMax}."); // El máximo es 10.
}
```

Tu puedes usar pattern matching para lograr el mismo comportamiento en Rust:

Sería más conciso usar `if let`:

```
let max = Some(10u32);
if let Some(max) = max {
    println!("El máximo es {}.", max); // El máximo es 10.
}
```

Operadores de Condición Nula

Los operadores null-condicionales (`?.` y `?[]`) facilitan el manejo de null en C#. En Rust, es mejor reemplazarlos usando el método `map` . El siguiente fragmento muestra la comparación:

```
string? some = "Hola, Mundo!";
string? none = null;
Console.WriteLine(some?.Length); // 12
Console.WriteLine(none?.Length); // (blank)
```

```
let some: Option<String> = Some(String::from("Hola, Mundo!"));
let none: Option<String> = None;
println!("{:?}", some.map(|s| s.len())); // Some(12)
println!("{:?}", none.map(|s| s.len())); // None
```

Null-coalescing operator

El null-coalescing operator (`??`) es típicamente usado para por defecto usar otro valor cuando un nulleable es `null` :

```
int? some = 1;
int? none = null;
Console.WriteLine(some ?? 0); // 1
Console.WriteLine(none ?? 0); // 0
```

En Rust, tu puedes usar `unwrap_or` para obtener el mismo comportamiento:

```
let some: Option<i32> = Some(1);
let none: Option<i32> = None;
println!("{:?}", some.unwrap_or(0)); // 1
println!("{:?}", none.unwrap_or(0)); // 0
```

Nota: Si el valor por defecto es costoso para computar, tu puedes usar `unwrap_or_else` en su lugar. Este tomara una closure como argumento, la cual permitirá inicializar un valor por defecto de forma perezosa.

Null-forgiving operator

El operador null-forgiving (`!`) no corresponde a un equivalente construido en Rust, como este solo afecta al flujo de análisis estático en el compilador de C#. En Rust, esto no es

necesario de usar para un sustituto de este.

Descartes

En C#, los descartes expresan al compilador y a otros que ignoren los resultados (o partes) de una expresión.

Hay múltiples contextos donde se pueden aplicar, por ejemplo, como un ejemplo básico, para ignorar el resultado de una expresión. En C#, se vería así:

```
_ = city.GetCityInformation(cityName);
```

En Rust, ignorar el resultado de una expresión se ve de manera idéntica:

```
_ = city.get_city_information(city_name);
```

Los descartes también se aplican para la destrucción de tuplas en C#:

```
var (_, second) = ("first", "second");
```

y, de manera idéntica, en Rust:

```
let (_, second) = ("first", "second");
```

Además de la destrucción de tuplas, Rust ofrece desestructuración de estructuras y enumeraciones usando `..`, donde `..` representa la parte restante de un tipo:

```
struct Point {
    x: i32,
    y: i32,
    z: i32,
}

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x), // x is 0
}
```

Cuando se realiza pattern matching, a menudo es útil descartar o ignorar parte de una expresión coincidente, por ejemplo, en C#:

```
_ = ("first", "second") switch
{
    ("first", _) => "first element matched",
    (_, _) => "first element did not match"
};
```

Y nuevamente, esto se ve casi idéntico en Rust:

```
_ = match ("first", "second")
{
    ("first", _) => "first element matched",
    (_, _) => "first element did not match"
};
```

Conversión y Casting

Ambos en C# y Rust son estéticamente tipados en tiempo de compilación. Por eso, luego que una variable es declarada, asignar un valor de un tipo diferente (A menos que este sea implícitamente convertible a el tipo esperado) de la variable esta prohibido. Hay multiples formas para convertir tipos en C# que equivalen a Rust.

Conversiones implícitas

Las conversiones implícitas existen en C# como en Rust (llamadas cohesiones de tipos). Considera el siguiente ejemplo:

```
int intNumber = 1;
long longNumber = intNumber;
```

Rust es mucho más restrictivo al respecto de cual cohesion de tipo permitir:

```
let int_number: i32 = 1;
let long_number: i64 = int_number; // error: expected `i64`, found `i32`
```

Un ejemplo para una conversión implícita valida usando subtipificación es:

```
fn bar<'a>() {
    let s: &'static str = "hi";
    let t: &'a str = s;
}
```

Mirar también:

- cohesion Deref
- Subtipificación y varianza

Conversión Explicita

Si convertir puede causar perdida de información, C# require conversión explicita usando una expresión de casting:

```
double a = 1.2;
int b = (int)a;
```

Conversiones explicitas pueden potencialmente fallar durante tiempo de ejecución con excepciones como `OverflowException` o `InvalidCastException` cuando se hace *down-casting*.

Rust no provee cohesión entre tipos primitivos, pero en su lugar usa conversion explicita usando la palabra clave `as` (casting). Usar Casting en Rust no causara pánico.

```
let int_number: i32 = 1;
let long_number: i64 = int_number as _;
```

Conversión Personalizada

Comúnmente, los tipos de .Net proveen operadores de conversión definidos por el usuario para convertir un tipo a otro tipo. También, `System.IConvertible` tiene el propósito de convertir un tipo en otro.

En Rust, la librería estándar contiene una abstracción para convertir un valor en un tipo diferente, con el trait `From` y recíprocamente `Into`. Cuando implementas `From` para un tipo, una implementación por default de `Into` es automáticamente provista (A esto se le llama *blanket implementation* en Rust). El siguiente ejemplo ilustra dos conversiones de tipos.

```
fn main() {
    let my_id = MyId("id".into()); // `into()` es implementado automáticamente
debido a la implementación del `From<&str>` trait para `String`.
    println!("{}", String::from(my_id)); // Esto usa la implementación
`From<MyId>` de `String`.
}

struct MyId(String);

impl From<MyId> for String {
    fn from(MyId(value): MyId) -> Self {
        value
    }
}
```

Mirar también:

- `TryFrom` y `TryInto` para versiones de `From` y `Into` que puede fallar.

Sobrecarga de operadores

Un tipo personalizado puede sobrecargar un *operador sobrecargable* en C#. Considera el siguiente ejemplo en C#:

```
Console.WriteLine(new Fraction(5, 4) + new Fraction(1, 2));  // 14/8

public readonly record struct Fraction(int Numerator, int Denominator)
{
    public static Fraction operator +(Fraction a, Fraction b) =>
        new(a.Numerator * b.Denominator + b.Numerator * a.Denominator,
a.Denominator * b.Denominator);

    public override string ToString() => $"{Numerator}/{Denominator}";
}
```

En Rust, muchos operadores pueden sobrecargarse mediante traits. Esto es posible porque los operadores son azúcar sintáctica para llamadas a métodos. Por ejemplo, el operador `+` en `a + b` llama al método `add` (ver sobrecarga de operadores):

```rust
use std::{fmt::{Display, Formatter, Result}, ops::Add};

struct Fraction {
    numerator: i32,
    denominator: i32,
}

impl Display for Fraction {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result {
        f.write_fmt(format_args!("{}/{}", self.numerator, self.denominator))
    }
}

impl Add<Fraction> for Fraction {
    type Output = Fraction;

    fn add(self, rhs: Fraction) -> Fraction {
        Fraction {
            numerator: self.numerator * rhs.denominator + rhs.numerator *
self.denominator,
            denominator: self.denominator * rhs.denominator,
        }
    }
}

fn main() {
    println!(
        "{}",
        Fraction { numerator: 5, denominator: 4 } + Fraction { numerator: 1,
denominator: 2 }
    ); // 14/8
}
```

Comentarios de Documentación

C# provee un mecanismo para documentar las APIs para tipos usando la sintaxis de un comentario que contiene texto XML. El compilador de C# produce un archivo XML que contiene estructuras de datos representando el comentario y la firma de la API. Otras herramientas pueden procesar la salida para proveer documentación legible para humanos en una forma diferente. Un ejemplo simple en C#:

```
/// <summary>
/// Esto es un comentario para documentar <c>MyClass</c>.
/// </summary>
public class MyClass {}
```

En Rust, los comentarios de documentación proporcionan el equivalente a los comentarios de documentación de C#. Los comentarios de documentación en Rust utilizan la sintaxis de Markdown. rustdoc es el compilador de documentación para el código Rust y generalmente se invoca a través de cargo doc, que compila los comentarios en documentación. Por ejemplo:

```
/// Este es un comentario de documentación para `MyStruct`.
struct MyStruct;
```

En el .Net SDK hay equivalente a `cargo doc`, como `dotnet doc`.

Mira también:

- How to write documentation
- Documentation tests

Gestión de Memoria

Al igual que C# y .NET, Rust tiene *memoria segura* para evitar toda clase de errores relacionados con el acceso a la memoria, que terminan siendo la fuente de muchas vulnerabilidades de seguridad en el software. Sin embargo, Rust puede garantizar la seguridad de memoria en tiempo de compilación; no hay una verificación en tiempo de ejecución (como el CLR). La única excepción aquí son las verificaciones de límites de arreglos que realiza el código compilado en tiempo de ejecución, ya sea el compilador de Rust o el compilador JIT en .NET. Al igual que en C#, también es posible escribir código inseguro en Rust, y de hecho, ambos lenguajes incluso comparten la misma palabra clave, *literalmente* `unsafe`, para marcar funciones y bloques de código donde ya no se garantiza la seguridad de memoria.

Rust no tiene un garbage collector (GC). Toda la gestión de memoria es completamente responsabilidad del desarrollador. Dicho esto, *Rust seguro* tiene reglas rodeando el concepto de *ownership* que aseguran que la memoria se libere *tan pronto como* ya no esté en uso (por ejemplo, al salir del ámbito de un bloque o de una función). El compilador hace un trabajo tremendo, a través del análisis estático en tiempo de compilación, para ayudar a gestionar esa memoria mediante las reglas de ownership. Si se violan, el compilador rechaza el código con un error de compilación.

En .NET, no existe el concepto de ownership de la memoria más allá de las raíces del Gargabe Collector (campos estáticos, variables locales en la pila de un hilo, registros de la CPU, manejadores, etc.). Es el GC quien recorre desde las raíces durante una recolección para determinar toda la memoria en uso siguiendo las referencias y purgando el resto. Al diseñar tipos y escribir código, un desarrollador de .NET puede permanecer ajeno al ownership, la gestión de memoria e incluso al funcionamiento del recolector de basura en su mayor parte, excepto cuando el código sensible al rendimiento requiere prestar atención a la cantidad y la velocidad a la que se asignan objetos en el montón. En contraste, las reglas del ownership de Rust requieren que el desarrollador piense y exprese explícitamente la propiedad en todo momento y esto impacta todo, desde el diseño de funciones, tipos, estructuras de datos hasta la forma en que se escribe el código. Además de eso, Rust tiene reglas estrictas sobre cómo se utiliza la información, de tal manera que puede identificar en tiempo de compilación data race conditions, así como problemas de corrupción (requiriendo seguridad en hilos) que podrían ocurrir potencialmente en tiempo de ejecución. Esta sección solo se enfocará en la gestión de memoria y la propiedad.

En Rust, solo puede haber un propietario de una porción de memoria, ya sea en el stack o en el heap, respaldando una estructura en un momento dado. El compilador asigna lifetimes y rastrea el ownership. Es posible pasar o ceder el ownership, lo cual se denomina *mover* en Rust. Estas ideas se ilustran brevemente en el siguiente código de ejemplo de Rust:

```
#![allow(dead_code, unused_variables)]

struct Point {
    x: i32,
    y: i32,
}

fn main() {
  let a = Point { x: 12, y: 34 }; // La instancia de Point es propiedad de a
  let b = a;                      // ahora b es propietario de la instancia de
Point
  println!("{}, {}", a.x, a.y);   // ¡error de compilación!
}
```

La primera instrucción en `main` asignará un `Point` y esa memoria será propiedad de `a`. En la segunda instrucción, la propiedad se mueve de `a` a `b` y `a` ya no puede ser utilizado porque ya no posee nada ni representa una memoria válida. La última instrucción que intenta imprimir los campos del punto a través de `a` fallará en la compilación. Supongamos que `main` se corrige para leerse de la siguiente manera:

```
fn main() {
    let a = Point { x: 12, y: 34 }; // La instancia de Point es propiedad de a
    let b = a;                      // ahora b es propietario de la instancia
de Point
    println!("{}, {}", b.x, b.y);   // ok, usamos b
}   // Point de b es liberado
```

Nota que cuando `main` termina, `a` y `b` saldrán de su ámbito. La memoria detrás de `b` será liberada en virtud de que la pila regresará a su estado previo a la llamada de `main`. En Rust, se dice que el punto detrás de `b` fue *descartado*. Sin embargo, dado que `a` cedió su propiedad del punto a `b`, no hay nada que descartar cuando `a` sale de su ámbito.

Una `struct` en Rust puede definir el código a ejecutar cuando se descarta una instancia implementando el trait `Drop`.

El equivalente aproximado de *dropping* en C# sería un finalizador de clase, pero mientras que un finalizador es llamado *automáticamente* por el GC en algún momento futuro, el *dropping* en Rust siempre es instantáneo y determinista; es decir, ocurre en el punto en que el compilador ha determinado que una instancia no tiene propietario basándose en los ámbitos y los lifetimes. En .NET, el equivalente de `Drop` sería `IDisposable` y se implementa en tipos para liberar cualquier recurso no administrado o memoria que posean. La *disposición determinística* no está impuesta ni garantizada, pero la declaración `using` en C# se utiliza típicamente para delimitar el ámbito de una instancia de un tipo desechable de manera que se disponga de manera determinista, al final del bloque de la declaración

`using`.

Rust tiene la noción de un lifetime global denotada por `'static`, que es un especificador de lifetime reservado. Una aproximación muy general en C# serían los campos estáticos de solo lectura de los tipos.

En C# y .NET, las referencias se comparten libremente sin mucha consideración, por lo que la idea de un único propietario y ceder/mover la propiedad puede parecer muy limitante en Rust, pero es posible tener *propiedad compartida* en Rust utilizando el tipo de puntero inteligente `Rc`; añade un conteo de referencias. Cada vez que el puntero inteligente es clonado, se incrementa el conteo de referencias. Cuando el clon se descarta, el conteo de referencias se decrementa. La instancia real detrás del puntero inteligente se descarta cuando el conteo de referencias alcanza cero. Estos puntos se ilustran mediante los siguientes ejemplos que se basan en los anteriores:

```
#![allow(dead_code, unused_variables)]

use std::rc::Rc;

struct Point {
    x: i32,
    y: i32,
}

impl Drop for Point {
    fn drop(&mut self) {
        println!("¡Point descartado!");
    }
}

fn main() {
    let a = Rc::new(Point { x: 12, y: 34 });
    let b = Rc::clone(&a); // compartido con b
    println!("a = {}, {}", a.x, a.y); // esta bien usar a
    println!("b = {}, {}", b.x, b.y); // y b
}

// imprime:
// a = 12, 34
// b = 12, 34
// ¡Point descartado!
```

Ten en cuenta que:

- `Point` implementa el método `drop` del trait `Drop` e imprime un mensaje cuando se descarta una instancia de `Point`.

- El punto creado en `main` está envuelto detrás del puntero inteligente `Rc`, por lo que el puntero inteligente *posee* el punto y no `a`.
- `b` obtiene un clon del puntero inteligente que efectivamente incrementa el conteo de referencias a 2. A diferencia del ejemplo anterior, donde `a` transfirió la propiedad del punto a `b`, tanto `a` como `b` poseen sus propios clones distintos del puntero inteligente, por lo que está bien seguir usando `a` y `b`.
- El compilador habrá determinado que `a` y `b` salen de su ámbito al final de `main` y por lo tanto inyectará llamadas para descartar cada uno. La implementación de `Drop` de `Rc` decrementará el conteo de referencias y también descartará lo que posee si el conteo de referencias ha alcanzado cero. Cuando eso sucede, la implementación de `Drop` de `Point` imprimirá el mensaje, "¡Point descartado!". El hecho de que el mensaje se imprima una vez demuestra que solo se creó, compartió y descartó un punto.

`Rc` no es seguro para hilos. Para la propiedad compartida en un programa multiproceso, la biblioteca estándar de Rust ofrece `Arc` en su lugar. El lenguaje Rust evitará el uso de `Rc` entre hilos.

En .NET, los tipos de valor (como `enum` y `struct` en C#) residen en el stack y los tipos de referencia (`interface`, `record class` y `class` en C#) se asignan en el heap. En Rust, el tipo de tipo (básicamente `enum` o `struct` *en Rust*) no determina dónde residirá finalmente la memoria de respaldo. Por defecto, siempre está en el stack, pero al igual que .NET y C# tienen la noción de hacer boxing de los tipos de valor, lo que los copia al heap, la forma de asignar un tipo en el heap es hacer boxing usando `Box`:

```
let stack_point = Point { x: 12, y: 34 };
let heap_point = Box::new(Point { x: 12, y: 34 });
```

Al igual que `Rc` y `Arc`, `Box` es un puntero inteligente, pero a diferencia de `Rc` y `Arc`, posee exclusivamente la instancia detrás de él. Todos estos punteros inteligentes asignan una instancia de su argumento de tipo `T` en el heap.

La palabra clave `new` en C# crea una instancia de un tipo, y aunque miembros como `Box::new` y `Rc::new` que ves en los ejemplos pueden parecer tener un propósito similar, `new` no tiene una designación especial en Rust. Es simplemente un *nombre convencional* que se utiliza para denotar un factory. De hecho, se les llama *funciones asociadas* del tipo, que es la manera de Rust de decir métodos estáticos.

Administración de Recursos

En la sección anterior, Administración de Memoria explicamos la diferencia entre .NET y Rust cuando se trata del Garbage Collector, Ownership y finalizadores. Esto is altamente recomendado para leer.

Esta sección es limitada a proporcionar un ejemplo de una conexión de base de datos ficticia que involucra una conexión SQL que debe cerrarse/disposed/destruirse adecuadamente.

```
{
    using var db1 = new DatabaseConnection("Server=A;Database=DB1");
    using var db2 = new DatabaseConnection("Server=A;Database=DB2");

    // ...código usando "db1" y "db2"...
}   // "Dispose" de "db1" y "db2" se llamabara aquí; cuando su scope termine

public class DatabaseConnection : IDisposable
{
    readonly string connectionString;
    SqlConnection connection; //Esto implementa IDisposable

    public DatabaseConnection(string connectionString) =>
        this.connectionString = connectionString;

    public void Dispose()
    {
        //Asegurando que se desecha la SqlConnection
        this.connection.Dispose();
        Console.WriteLine("Closing connection: {this.connectionString}");
    }
}
```

```
struct DatabaseConnection(&'static str);

impl DatabaseConnection {
    // ...funciones para usar la conexión de base de datos...
}

impl Drop for DatabaseConnection {
    fn drop(&mut self) {
        // ...cerrando la conexión...
        self.close_connection();
        // ...imprimiendo un mensaje...
        println!("Cerrando conexión: {}", self.0)
    }
}

fn main() {
    let _db1 = DatabaseConnection("Server=A;Database=DB1");
    let _db2 = DatabaseConnection("Server=A;Database=DB2");
    // ...code for making use of the database connection...
    // ...codigo para utilizar la conexión a la base de datos...
}   // "Dispose" de "db1" y "db2" se llamabara aquí; cuando su scope termine
```

En .NET, intentar usar un objeto después de llamar a `Dispose` en él típicamente causará que se lance una `ObjectDisposedException` en tiempo de ejecución. En Rust, el compilador garantiza en tiempo de compilación que esto no puede suceder.

Threading

La biblioteca estándar de Rust admite hilos, sincronización y concurrencia. Aunque el lenguaje en sí y la biblioteca estándar tienen soporte básico para estos conceptos, gran parte de la funcionalidad adicional es proporcionada por crates y no se cubrirá en este documento.

A continuación se presenta una lista aproximada de la correspondencia entre los tipos y métodos de hilos en .NET y Rust:

.NET	Rust
`Thread`	`std::thread::thread`
`Thread.Start`	`std::thread::spawn`
`Thread.Join`	`std::thread::JoinHandle`
`Thread.Sleep`	`std::thread::sleep`
`ThreadPool`	-
`Mutex`	`std::sync::Mutex`
`Semaphore`	-
`Monitor`	`std::sync::Mutex`
`ReaderWriterLock`	`std::sync::RwLock`
`AutoResetEvent`	`std::sync::Condvar`
`ManualResetEvent`	`std::sync::Condvar`
`Barrier`	`std::sync::Barrier`
`CountdownEvent`	`std::sync::Barrier`
`Interlocked`	`std::sync::atomic`
`Volatile`	`std::sync::atomic`
`ThreadLocal`	`std::thread_local`

Lanzar un hilo y esperar a que termine funciona de la misma manera en C#/.NET y Rust. A continuación, se muestra un programa simple en C# que crea un hilo (donde el hilo imprime algún texto en la salida estándar) y luego espera a que termine:

```
using System;
using System.Threading;

var thread = new Thread(() => Console.WriteLine("¡Hola, desde un hilo!"));
thread.Start();
thread.Join(); // espera a que el hilo termine
```

El mismo código en Rust sería el siguiente:

```
use std::thread;

fn main() {
    let thread = thread::spawn(|| println!("¡Hola, desde un hilo!"));
    thread.join().unwrap(); // espera a que el hilo termine
}
```

Crear e inicializar un objeto hilo y comenzar un hilo son dos acciones diferentes en .NET, mientras que en Rust ambas ocurren al mismo tiempo con `thread::spawn`.

En .NET, es posible enviar datos como un argumento a un hilo:

```
#nullable enable

using System;
using System.Text;
using System.Threading;

var t = new Thread(obj =>
{
    var data = (StringBuilder)obj!;
    data.Append(" Mundo!");
});

var data = new StringBuilder("¡Hola");
t.Start(data);
t.Join();

Console.WriteLine($"Frase: {data}");
```

Sin embargo, una versión más moderna o concisa usaría closures:

```
using System;
using System.Text;
using System.Threading;

var data = new StringBuilder("¡Hola");

var t = new Thread(obj => data.Append(" Mundo!"));

t.Start();
t.Join();

Console.WriteLine($"Frase: {data}");
```

En Rust, no hay ninguna variante de `thread::spawn` que haga lo mismo. En su lugar, los

datos se pasan al hilo mediante un cierre closure:

```
use std::thread;

fn main() {
    let data = String::from("¡Hola");
    let handle = thread::spawn(move || {
        let mut data = data;
        data.push_str(" Mundo!");
        data
    });
    println!("Frase: {}", handle.join().unwrap());
}
```

Algunas cosas a tener en cuenta:

- La palabra clave `move` es *necesaria* para *mover* o pasar la propiedad de `data` al cierre para el hilo. Una vez hecho esto, ya no es legal seguir utilizando la variable `data` en `main`. Si es necesario, `data` debe ser copiada o clonada (dependiendo de lo que admita el tipo de valor).
- Los hilos de Rust pueden devolver valores, como las tareas en C#, lo que se convierte en el valor de retorno del método `join`.
- Es posible también pasar datos al hilo de C# mediante una closure, como en el ejemplo de Rust, pero la versión de C# no necesita preocuparse por el ownership ya que la memoria detrás de los datos será reclamada por el GC una vez que nadie la esté referenciando.

Sincronización

Cuando los datos son compartidos entre hilos, es necesario sincronizar el acceso de lectura y escritura a los datos para evitar la corrupción. En C#, se ofrece la palabra clave `lock` como un primitivo de sincronización (que se desenrolla en el uso seguro de excepciones de `Monitor` de .NET):

```
using System;
using System.Threading;

var dataLock = new object();
var data = 0;
var threads = new List<Thread>();

for (var i = 0; i < 10; i++)
{
    var thread = new Thread(() =>
    {
        for (var j = 0; j < 1000; j++)
        {
            lock (dataLock)
                data++;
        }
    });
    threads.Add(thread);
    thread.Start();
}

foreach (var thread in threads)
    thread.Join();

Console.WriteLine(data);
```

En Rust, uno debe hacer uso explícito de estructuras de concurrencia como `Mutex`:

```rust
use std::thread;
use std::sync::{Arc, Mutex};

fn main() {
    let data = Arc::new(Mutex::new(0)); // (1)

    let mut threads = vec![];
    for _ in 0..10 {
        let data = Arc::clone(&data); // (2)
        let thread = thread::spawn(move || { // (3)
            for _ in 0..1000 {
                let mut data = data.lock().unwrap();
                *data += 1; // (4)
            }
        });
        threads.push(thread);
    }

    for thread in threads {
        thread.join().unwrap();
    }

    println!("{}", data.lock().unwrap());
}
```

Algunas cosas a tener en cuenta:

- Dado que la propiedad de la instancia de `Mutex` y, a su vez, los datos que protege serán compartidos por múltiples hilos, se envuelve en un `Arc` (1). `Arc` proporciona recuento de referencias atómico, que se incrementa cada vez que se clona (2) y se decrementa cada vez que se elimina. Cuando el recuento alcanza cero, se elimina el mutex y, por lo tanto, los datos que protege. Esto se discute con más detalle en Gestión de Memoria.
- La closure para cada hilo recibe la propiedad (3) de la *referencia clonada* (2).
- El código similar a un puntero, que es `*data += 1` (4), no es un acceso a puntero inseguro incluso si parece serlo. Está actualizando los datos *envueltos* en el mutex guard.

A diferencia de la versión de C#, donde se puede volver inseguro para hilos al comentar la declaración `lock` , la versión de Rust se negará a compilar si se cambia de alguna manera (por ejemplo, al comentar partes) que la vuelva insegura para hilos. Esto demuestra que escribir código seguro para hilos es responsabilidad del desarrollador en C# y .NET mediante el uso cuidadoso de estructuras sincronizadas, mientras que en Rust, uno puede confiar en el compilador.

El compilador puede ayudar porque las estructuras de datos en Rust están marcadas por *traits* especiales (ver Interfaces): `Sync` y `Send`. `Sync` indica que las referencias a las instancias de un tipo son seguras para compartir entre hilos. `Send` indica que es seguro enviar instancias de un tipo a través de los límites de los hilos. Para obtener más información, consulta el capítulo "Concurrencia sin miedo" del libro de Rust.

Productor-Consumidor

El patrón productor-consumidor es muy común para distribuir trabajo entre hilos donde los datos son pasados desde hilos productores a hilos consumidores sin necesidad de compartir o bloquear. .NET tiene un soporte muy amplio para esto, pero en el nivel más básico, `System.Collections.Concurrent` proporciona `BlockingCollection` como se muestra en el siguiente ejemplo en C#:

```
using System;
using System.Threading;
using System.Collections.Concurrent;

var messages = new BlockingCollection<string>();
var producer = new Thread(() =>
{
    for (var n = 1; i < 10; i++)
        messages.Add($"Mensaje #{n}");
    messages.CompleteAdding();
});

producer.Start();

// el hilo principal es el consumidor aquí
foreach (var message in messages.GetConsumingEnumerable())
    Console.WriteLine(message);

producer.Join();
```

Lo mismo se puede hacer en Rust utilizando *canales*. La biblioteca estándar principalmente proporciona `mpsc::channel`, que es un canal que admite múltiples productores y un único consumidor. Una traducción aproximada del ejemplo anterior en C# a Rust se vería así:

```rust
use std::thread;
use std::sync::mpsc;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    let producer = thread::spawn(move || {
        for n in 1..10 {
            tx.send(format!("Mensaje #{}", n)).unwrap();
        }
    });

    // el hilo principal es el consumidor aquí
    for received in rx {
        println!("{}", received);
    }

    producer.join().unwrap();
}
```

Al igual que los canales en Rust, .NET también ofrece canales en el espacio de nombres `System.Threading.Channels`, pero está diseñado principalmente para ser utilizado con tareas y programación asincrónica mediante el uso de `async` y `await`. El equivalente de los canales amigables para asincronía en el espacio de Rust es ofrecido por el runtime de Tokio.

Testing

Organización de pruebas

Las soluciones .NET utilizan proyectos separados para alojar el código de prueba, independientemente del framework de pruebas utilizado (xUnit, NUnit, MSTest, etc.) y el tipo de pruebas (unitarias o de integración) que se estén escribiendo. Por lo tanto, el código de los test vive en un espacio separado al código de la aplicación o biblioteca que se está probando. En Rust, es mucho más convencional que las *pruebas unitarias* se encuentren en un submódulo de prueba separado (convencionalmente) llamado `tests`, pero que se coloca en el mismo *archivo fuente* que el código del módulo de aplicación o biblioteca que es objeto de las pruebas. Esto tiene dos beneficios:

- El código/módulo y sus pruebas unitarias viven juntos.
- No hay necesidad de un truco como `[InternalsVisibleTo]` que existe en .NET porque las pruebas tienen acceso a los elementos internos al ser un submódulo.

El submódulo de prueba está anotado con el atributo `#[cfg(test)]`, lo que tiene el efecto de que todo el módulo se compila (condicionalmente) y se ejecuta solo cuando se emite el comando `cargo test`.

Dentro de los submódulos de prueba, las funciones de prueba están anotadas con el atributo `#[test]`.

Las pruebas de integración suelen estar en un directorio llamado `tests` que se encuentra adyacente al directorio `src` con las pruebas unitarias y el código fuente. `cargo test` compila cada archivo en ese directorio como un crate separado y ejecuta todos los métodos anotados con el atributo `#[test]`. Dado que se entiende que las pruebas de integración están en el directorio `tests`, no es necesario marcar los módulos allí con el atributo `#[cfg(test)]`.

Mirar también:

- Test Organization

Ejecución de pruebas

Tan simple como puede ser, el equivalente de `dotnet test` en Rust es `cargo test`.

El comportamiento predeterminado de `cargo test` es ejecutar todas las pruebas en paralelo, pero esto se puede configurar para que se ejecuten de manera consecutiva utilizando solo un hilo:

```
cargo test -- --test-threads=1
```

Para obtener más información, consulta "Ejecutando tests en paralelo o consecutivamente".

ejecutando-tests-en-paralelo-o-consecutivamente

Output en las Pruebas

Para pruebas de integración o de extremo a extremo muy complejas, a veces los desarrolladores de .NET registran lo que está sucediendo durante una prueba. La forma en que hacen esto varía con cada framework de pruebas. Por ejemplo, en NUnit, esto es tan simple como usar `Console.WriteLine`, pero en XUnit, se utiliza `ITestOutputHelper`. En Rust, es similar a NUnit; es decir, simplemente se escribe en la salida estándar usando `println!`. La salida capturada durante la ejecución de las pruebas no se muestra por defecto a menos que `cargo test` se ejecute con la opción `--show-output`:

```
cargo test --show-output
```

Para obtener más información, consulta "Mostrando el Output de las funciones".

mostrando-el-output-de-las-funciones

Aserciones

Los usuarios de .NET tienen múltiples formas de hacer aserciones, dependiendo del framework de trabajo que estén utilizando. Por ejemplo, una aserción en xUnit.net podría lucir así:

```
[Fact]
public void Something_Is_The_Right_Length()
{
    var value = "something";
    Assert.Equal(9, value.Length);
}
```

Rust no requiere un framework o crate separado. La biblioteca estándar viene con *macros* integradas que son lo suficientemente buenas para la mayoría de las afirmaciones en las pruebas:

- `assert!`
- `assert_eq!`
- `assert_ne!`

A continuación se muestra un ejemplo de `assert_eq` en acción:

```
#[test]
fn something_is_the_right_length() {
    let value = "something";
    assert_eq!(9, value.len());
}
```

La biblioteca estándar no ofrece nada en la dirección de pruebas basadas en datos, como `[Theory]` en xUnit.net.

Mocking

Cuando se escriben pruebas para una aplicación o biblioteca .NET, existen varios framework, como Moq y NSubstitute, para simular las dependencias de los tipos. También hay crates similares para Rust, como `mockall`, que pueden ayudar con la simulación. Sin embargo, también es posible usar compilación condicional haciendo uso del atributo `cfg` como un medio simple para la simulación sin necesidad de depender de crates o frameworks externos. El atributo `cfg` incluye condicionalmente el código que anota en función de un símbolo de configuración, como `test` para pruebas. Esto no es muy diferente de usar `DEBUG` para compilar condicionalmente código específicamente para compilaciones de depuración. Una desventaja de este enfoque es que solo se puede tener una implementación para todas las pruebas del módulo.

Cuando se especifica, el atributo `#[cfg(test)]` le indica a Rust que compile y ejecute el código solo cuando se ejecute el comando `cargo test`, que ejecuta el compilador con

`rustc --test`. Lo contrario es cierto para el atributo `#[cfg(not(test))]`; incluye el código anotado solo cuando se realiza la prueba con `cargo test`.

El ejemplo a continuación muestra la simulación de una función independiente `var_os` de la biblioteca estándar que lee y devuelve el valor de una variable de entorno. Importa condicionalmente una versión simulada de la función `var_os` utilizada por `get_env`. Cuando se compila con `cargo build` o se ejecuta con `cargo run`, el binario compilado hará uso de `std::env::var_os`, pero `cargo test` en su lugar importará `tests::var_os_mock` como `var_os`, lo que hará que `get_env` utilice la versión simulada durante las pruebas:

```
// Derechos de autor (c) Microsoft Corporation. Todos los derechos reservados.
// Licenciado bajo la licencia MIT.

/// Función de utilidad para leer una variable de entorno y devolver su valor
/// si está definida. Falla/pániquea si el valor no es Unicode válido.
pub fn get_env(key: &str) -> Option<String> {
    #[cfg(not(test))]                // para compilaciones regulares...
    use std::env::var_os;            // ...importar desde la biblioteca
estándar
    #[cfg(test)]                     // para compilaciones de prueba...
    use tests::var_os_mock as var_os; // ...importar la simulación desde el
submódulo de prueba

    let val = var_os(key);
    val.map(|s| s.to_str()     // obtiene slice de string
                 .unwrap()     // lanza un pánico si no es Unicode válido
                 .to_owned())  // convierte a "String"
}

#[cfg(test)]
mod tests {
    use std::ffi::*;
    use super::*;

    pub(crate) fn var_os_mock(key: &str) -> Option<OsString> {
        match key {
            "FOO" => Some("BAR".into()),
            _ => None
        }
    }

    #[test]
    fn get_env_when_var_undefined_returns_none() {
        assert_eq!(None, get_env("???"));
    }

    #[test]
    fn get_env_when_var_defined_returns_some_value() {
        assert_eq!(Some("BAR".to_owned()), get_env("FOO"));
    }
}
```

Cobertura de código

Existe herramientas sofisticadas para .NET en cuanto a análisis de cobertura de código de pruebas. En Visual Studio, las herramientas están integradas de forma nativa. En Visual Studio Code, existen complementos. Los desarrolladores de .NET podrían estar

familiarizados con coverlet también.

Rust proporciona implementaciones integradas de cobertura de código para recopilar la cobertura de código de las pruebas.

También hay complementos disponibles para Rust que ayudan con el análisis de cobertura de código. No está integrado de manera perfecta, pero con algunos pasos manuales, los desarrolladores pueden analizar su código de manera visual.

La combinación del complemento Coverage Gutters para Visual Studio Code y Tarpaulin permite el análisis visual de la cobertura de código en Visual Studio Code. Coverage Gutters requiere un archivo LCOV. Se pueden usar otras herramientas además de Tarpaulin para generar ese archivo.

Una vez configurado, ejecuta el siguiente comando:

```
cargo tarpaulin --ignore-tests --out Lcov
```

Esto genera un archivo de cobertura de código LCOV. Una vez habilitado `Coverage Gutters: Watch`, será recogido por el complemento Coverage Gutters, que mostrará indicadores visuales en línea sobre la cobertura de líneas en el editor de código fuente.

> Nota: La ubicación del archivo LCOV es esencial. Si hay un espacio de trabajo (ver Estructura del Proyecto) con múltiples paquetes y se genera un archivo LCOV en la raíz usando `--workspace`, ese es el archivo que se está utilizando, incluso si hay un archivo presente directamente en la raíz del paquete. Es más rápido aislar el paquete específico bajo prueba en lugar de generar el archivo LCOV en la raíz.

Benchmarking

La ejecución de benchmarks en Rust se realiza a través de `cargo bench`, un comando específico para `cargo` que ejecuta todos los métodos anotados con el atributo `#[bench]`. Este atributo está actualmente inestable y disponible solo para el canal nightly.

Los usuarios de .NET pueden utilizar la biblioteca `BenchmarkDotNet` para realizar benchmarks de métodos y realizar un seguimiento de su rendimiento. El equivalente de `BenchmarkDotNet` es una crate llamada `Criterion`.

Según su documentación, `Criterion` recopila y almacena información estadística de ejecución en ejecución y puede detectar automáticamente regresiones de rendimiento, así como medir optimizaciones.

Usando `Criterion`, es posible utilizar el atributo `#[bench]` sin necesidad de cambiar al canal nightly.

Al igual que en `BenchmarkDotNet`, también es posible integrar los resultados de los benchmarks con la GitHub Action para Benchmarking Continuo. De hecho, `Criterion` admite múltiples formatos de salida, entre los que también se encuentra el formato `bencher`, que imita los benchmarks nightly de `libtest` y es compatible con la acción mencionada anteriormente.

Logging y Tracing

.NET admite varias API de logging. Para la mayoría de los casos, `ILogger` es una buena opción predeterminada, ya que funciona con una variedad de proveedores de registro integrados y de terceros. En C#, un ejemplo mínimo para registro estructurado podría lucir así:

```
using Microsoft.Extensions.Logging;

using var loggerFactory = LoggerFactory.Create(builder =>
builder.AddConsole());
var logger = loggerFactory.CreateLogger<Program>();
logger.LogInformation("Hola {Day}.", "Jueves"); // Hola Jueves.
```

En Rust, se proporciona una fachada de logging ligera a través de log. Tiene menos características que `ILogger`, por ejemplo, aún no ofrece (de manera estable) registro estructurado o ámbitos de registro.

Para algo con una paridad de características más cercana a .NET, Tokio ofrece `tracing`. `tracing` es un framework para instrumentar aplicaciones Rust para recopilar información de diagnóstico estructurada y basada en eventos. `tracing_subscriber` se puede utilizar para implementar y componer suscriptores de `tracing`. El mismo ejemplo de registro estructurado anterior con `tracing` y `tracing_subscriber` se vería así:

```
fn main() {
    // Instalar el recolector global de mensajes de ("consola").
    tracing_subscriber::fmt().init();
    tracing::info!("Hola {Day}.", Day = "Jueves"); // Hola Jueves.
}
```

OpenTelemetry ofrece una colección de herramientas, APIs y SDKs utilizados para instrumentar, generar, recopilar y exportar datos de telemetría basados en la especificación de OpenTelemetry. En el momento de escribir esto, la API de registro de OpenTelemetry aún no es estable y la implementación de Rust todavía no soporta el registro, pero sí soporta la API de rastreo.

Compilación Condicional

Tanto .NET como Rust proporcionan la posibilidad de compilar código específico basado en condiciones externas.

En .NET es posible utilizar algunas directivas del preprocesador para controlar la compilación condicional.

```
#if debug
    Console.WriteLine("Debug");
#else
    Console.WriteLine("No debug");
#endif
```

Además de los símbolos predefinidos, también es posible utilizar la opción del compilador *DefineConstants* para definir símbolos que se pueden utilizar con `#if`, `#else`, `#elif` y `#endif` para compilar archivos fuente de forma condicional en .NET.

En Rust, es posible utilizar el `atributo cfg`, el `atributo cfg_attr` o el `macro cfg` para controlar la compilación condicional.

Al igual que en .NET, además de los símbolos predefinidos, también es posible utilizar la bandera del compilador `--cfg` para establecer arbitrariamente opciones de configuración.

El `atributo cfg` requiere y evalúa un `ConfigurationPredicate`.

```
use std::fmt::{Display, Formatter};

struct MyStruct;

// Esta implementación de Display solo se incluye cuando el SO es Unix pero foo
no es igual a bar
// Puedes compilar un ejecutable para esta versión, en Linux, con 'rustc
main.rs --cfg foo=\"baz\"'
#[cfg(all(unix, not(foo = "bar")))]
impl Display for MyStruct {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        f.write_str("Ejecutando sin la configuración foo=bar")
    }
}

// Esta función solo se incluye cuando tanto unix como foo=bar están definidos
// Puedes compilar un ejecutable para esta versión, en Linux, con 'rustc
main.rs --cfg foo=\"bar\"'
#[cfg(all(unix, foo = "bar"))]
impl Display for MyStruct {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        f.write_str("Ejecutando con la configuración foo=bar")
    }
}

// Esta función provoca un pánico cuando no se compila para Unix
// Puedes compilar un ejecutable para esta versión, en Windows, con 'rustc
main.rs'
#[cfg(not(unix))]
impl Display for MyStruct {
    fn fmt(&self, _f: &mut Formatter<'_>) -> std::fmt::Result {
        panic!()
    }
}

fn main() {
    println!("{}", MyStruct);
}
```

El `atributo cfg_attr` incluye condicionalmente atributos basados en un predicado de configuración.

```
#[cfg_attr(feature = "serialization_support", derive(Serialize, Deserialize))]
pub struct MaybeSerializableStruct;

// Cuando la feature flag `serialization_support` está habilitada, lo anterior
se expandirá a:
// #[derive(Serialize, Deserialize)]
// pub struct MaybeSerializableStruct;
```

The built-in `cfg macro` takes in a single configuration predicate and evaluates to the true literal when the predicate is true and the false literal when it is false.

El `macro cfg` incorporado toma un solo predicado de configuración y evalúa al literal verdadero cuando el predicado es verdadero y al literal falso cuando es falso.

```
if cfg!(unix) {
  println!("¡Estoy ejecutándome en una máquina Unix!");
}
```

Mira también:

- Conditional compilation

Features

La compilación condicional también es útil cuando es necesario proporcionar dependencias opcionales. Con las "features" de Cargo, un paquete define un conjunto de funcionalidades nombradas en la tabla `[features]` de Cargo.toml, y cada funcionalidad puede estar habilitada o deshabilitada. Las funcionalidades del paquete que se está construyendo pueden habilitarse en la línea de comandos con banderas como `--features` . Las funcionalidades para las dependencias pueden habilitarse en la declaración de dependencia en Cargo.toml.

Mira también:

- Features

Entorno y Configuración

Accediendo a variables de entorno

.NET proporciona acceso a las variables de entorno a través del método `System.Environment.GetEnvironmentVariable`. Este método recupera el valor de una variable de entorno en tiempo de ejecución.

```
using System;

const string name = "VARIABLE_EJEMPLO";

var value = Environment.GetEnvironmentVariable(name);
if (string.IsNullOrEmpty(value))
    Console.WriteLine($"Variable '{name}' no esta configurada.");
else
    Console.WriteLine($"Variable '{name}' configurada con '{value}'.");
```

Rust proporciona la misma funcionalidad de acceso a una variable de entorno en tiempo de ejecución mediante las funciones `var` y `var_os` del módulo `std::env`.

La función `var` devuelve un `Result<String, VarError>`, devolviendo la variable si está configurada o devolviendo un error si la variable no está configurada o no es Unicode válido.

`var_os` tiene una firma diferente, devolviendo una `Option<OsString>`, devolviendo algún valor si la variable está configurada o devolviendo None si la variable no está configurada. Un `OsString` no tiene que ser Unicode válido.

```
use std::env;


fn main() {
    let key = "VariableEjemplo";
    match env::var(key) {
        Ok(val) => println!("{key}: {val:?}"),
        Err(e) => println!("No se pudo interpretar {key}: {e}"),
    }
}
```

```rust
use std::env;

fn main() {
    let key = "VariableEjemplo";
    match env::var_os(key) {
        Some(val) => println!("{key}: {val:?}"),
        None => println!("{key} no definida en el entorno"),
    }
}
```

Rust también proporciona la funcionalidad de acceder a una variable de entorno en tiempo de compilación. El macro `env!` del módulo `std::env` expande el valor de la variable en tiempo de compilación, devolviendo un `&'static str`. Si la variable no está establecida, se emite un error.

```rust
use std::env;

fn main() {
    let example = env!("VariableEjemplo");
    println!("{example}");
}
```

En .NET, el acceso a variables de entorno en tiempo de compilación se puede lograr, de una manera menos directa, a través de generadores de código fuente.

Configuración

La configuración en .NET es posible mediante proveedores de configuración. El framework proporciona varias implementaciones de proveedores a través del espacio de nombres `Microsoft.Extensions.Configuration` y paquetes NuGet.

Los proveedores de configuración leen datos de configuración a partir de pares clave-valor utilizando diferentes fuentes y proporcionan una vista unificada de la configuración a través del tipo `IConfiguration`.

```
using Microsoft.Extensions.Configuration;

class Example {
    static void Main()
    {
        IConfiguration configuration = new ConfigurationBuilder()
            .AddEnvironmentVariables()
            .Build();

        var example = configuration.GetValue<string>("VariableEjemplo");

        Console.WriteLine(example);
    }
}
```

Otros ejemplos de proveedores se pueden encontrar en la documentación oficial Proveedores de configuración en .NET.

Una experiencia de configuración similar en Rust está disponible mediante el uso de crates de terceros como figment o config.

Vea el siguiente ejemplo utilizando el crate config:

```
use config::{Config, Environment};

fn main() {
    let builder = Config::builder().add_source(Environment::default());

    match builder.build() {
        Ok(config) => {
            match config.get_string("variable_ejemplo") {
                Ok(v) => println!("{v}"),
                Err(e) => println!("{e}")
            }
        },
        Err(_) => {
            // algo salio mal
        }
    }
}
```

LINQ

Esta sección discute LINQ en el contexto y con el propósito de consultar o transformar secuencias (`IEnumerable` / `IEnumerable<T>`) y, típicamente, colecciones como listas, conjuntos y diccionarios.

`IEnumerable<T>`

El equivalente de `IEnumerable<T>` en Rust es `IntoIterator` . Así como una implementación de `IEnumerable<T>.GetEnumerator()` devuelve un `IEnumerator<T>` en .NET, una implementación de `IntoIterator::into_iter` devuelve un `Iterator` . Sin embargo, cuando es momento de iterar sobre los elementos de un contenedor que anuncia soporte para la iteración a través de estos tipos, ambos lenguajes ofrecen azúcar sintáctica en forma de constructos de bucles para iterables. En C#, existe `foreach` :

```
using System;
using System.Text;

var values = new[] { 1, 2, 3, 4, 5 };
var output = new StringBuilder();

foreach (var value in values)
{
    if (output.Length > 0)
        output.Append(", ");
    output.Append(value);
}

Console.Write(output); // Imprime: 1, 2, 3, 4, 5
```

En Rust, el equivalente es simplemente `for` :

```
use std::fmt::Write;

fn main() {
    let values = [1, 2, 3, 4, 5];
    let mut output = String::new();

    for value in values {
        if output.len() > 0 {
            output.push_str(", ");
        }
        // ! descarta/ignora cualquier error de  write
        _ = write!(output, "{value}");
    }

    println!("{output}");  // Imprime: 1, 2, 3, 4, 5
}
```

El bucle `for` sobre un iterable esencialmente se descompone en lo siguiente:

```
use std::fmt::Write;

fn main() {
    let values = [1, 2, 3, 4, 5];
    let mut output = String::new();

    let mut iter = values.into_iter();      // obtiene el iterador
    while let Some(value) = iter.next() {   // en bucle mientras haya más
elementos
        if output.len() > 0 {
            output.push_str(", ");
        }
        _ = write!(output, "{value}");
    }

    println!("{output}");
}
```

Las reglas de ownership y data race conditions de Rust se aplican a todas las instancias y datos, y la iteración no es una excepción. Entonces, aunque iterar sobre un arreglo pueda parecer sencillo y muy similar a C#, hay que tener en cuenta la propiedad cuando se necesita iterar sobre la misma colección/iterable más de una vez. El siguiente ejemplo itera la lista de enteros dos veces: una vez para imprimir su suma y otra para determinar e imprimir el entero máximo:

```
fn main() {
    let values = vec![1, 2, 3, 4, 5];

    // suma todos los valores

    let mut sum = 0;
    for value in values {
        sum += value;
    }
    println!("sum = {sum}");

    // determina el valor maximo

    let mut max = None;
    for value in values {
        if let Some(some_max) = max { // si el máximo está definido
            if value > some_max {     // y el valor es mayor
                max = Some(value)     // entonces tenemos un nuevo máximo
            }
        } else {                      // el máximo es indefinido cuando la
interacción arranca
            max = Some(value)         // entonces establece el primer valor
como máximo
        }
    }
    println!("max = {max:?}");
}
```

Sin embargo, el código anterior es rechazado por el compilador debido a una diferencia sutil: `values` ha sido cambiado de un arreglo a un `Vec<int>` , un *vector*, que es el tipo de Rust para arreglos dinámicos (similar a `List<T>` en .NET). La primera iteración de `values` termina *consumiendo* cada valor a medida que se suman los enteros. En otras palabras, la propiedad de *cada elemento* en el vector pasa a la variable de iteración del bucle: `value` . Dado que `value` sale del alcance al final de cada iteración del bucle, la instancia que posee se elimina. Si `values` hubiera sido un vector de datos alojados en el heap, la memoria en el heap que respalda cada elemento se liberaría a medida que el bucle avanzara al siguiente elemento. Para solucionar el problema, uno debe solicitar la iteración sobre *referencias compartidas* usando `&values` en el bucle `for` . Como resultado, `value` será una referencia compartida a un elemento en lugar de tomar su propiedad.

A continuación se muestra la versión actualizada del ejemplo anterior que compila. La corrección consiste simplemente en reemplazar `values` por `&values` en cada uno de los bucles `for` .

```
fn main() {
    let values = vec![1, 2, 3, 4, 5];

    // suma todos los valores

    let mut sum = 0;
    for value in &values {
        sum += value;
    }
    println!("sum = {sum}");

    // determina el valor máximo

    let mut max = None;
    for value in &values {
        if let Some(some_max) = max { // si el máximo esta definido
            if value > some_max {     // y el valor es mayor
                max = Some(value)     // entonces tenemos un nuevo máximo
            }
        } else {                      // max no esta definido cuando empieza a
iterar
            max = Some(value)         // entonces asigna el primer valor
        }
    }
    println!("max = {max:?}");
}
```

El ownership y la liberación de recursos se pueden observar en acción incluso cuando `values` es un array en lugar de un vector. Considera solo el bucle de suma del ejemplo anterior sobre un array de una estructura que envuelve un entero:

```
struct Int(i32);

impl Drop for Int {
    fn drop(&mut self) {
        println!("{} liberado", self.0)
    }
}

fn main() {
    let values = [Int(1), Int(2), Int(3), Int(4), Int(5)];
    let mut sum = 0;

    for value in values {
        sum += value.0;
    }

    println!("sum = {sum}");
}
```

`Int` implementa `Drop` para que se imprima un mensaje cuando una instancia se libera. Al ejecutar el código anterior, se imprimirá:

```
value = Int(1)
Int(1) liberado
value = Int(2)
Int(2) liberado
value = Int(3)
Int(3) liberado
value = Int(4)
Int(4) liberado
value = Int(5)
Int(5) liberado
sum = 15
```

Es evidente que cada valor se adquiere y se libera mientras el bucle está en ejecución. Una vez que el bucle termina, se imprime la suma. Si `values` en el bucle `for` se cambia a `&values`, de esta forma:

```
for value in &values {
    // ...
}
```

entonces la salida del programa cambiará radicalmente:

```
value = Int(1)
value = Int(2)
value = Int(3)
value = Int(4)
value = Int(5)
sum = 15
Int(1) liberado
Int(2) liberado
Int(3) liberado
Int(4) liberado
Int(5) liberado
```

Esta vez, los valores se adquieren pero no se liberan durante el bucle porque cada elemento no es poseído por la variable del bucle de iteración. La suma se imprime una vez que el bucle termina. Finalmente, cuando el array `values`, que aún posee todas las instancias de `Int`, sale de alcance al final de `main`, su liberación, a su vez, libera todas las instancias de `Int`.

Estos ejemplos demuestran que, aunque iterar sobre tipos de colecciones puede parecer tener muchas similitudes entre Rust y C#, desde las construcciones de bucles hasta las abstracciones de iteración, aún existen diferencias sutiles con respecto a la propiedad que

pueden llevar al compilador a rechazar el código en algunos casos.

Mira también:

- Iterator
- Iterando por referencia

Operadores

Los *operadores* en LINQ están implementados en forma de métodos de extensión en C# que se pueden encadenar para formar un conjunto de operaciones, siendo lo más común la creación de una consulta sobre algún tipo de data source. C# también ofrece una *sintaxis de consulta* inspirada en SQL, con cláusulas como `from`, `where`, `select`, `join` y otras, que pueden servir como una alternativa o complemento al encadenamiento de métodos. Muchos bucles imperativos pueden reescribirse como consultas en LINQ, mucho más expresivas y componibles.

Rust no ofrece nada similar a la sintaxis de consultas de C#. Tiene métodos, llamados *[adaptadores]* en términos de Rust, sobre tipos iterables y, por lo tanto, directamente comparables al encadenamiento de métodos en C#. Sin embargo, mientras que reescribir un bucle imperativo como código LINQ en C# a menudo es beneficioso en términos de expresividad, robustez y componibilidad, existe un compromiso con el rendimiento. Los bucles imperativos orientados a cálculos *generalmente* se ejecutan más rápido porque el compilador JIT los puede optimizar y se incurren en menos despachos virtuales o invocaciones indirectas de funciones. Lo sorprendente en Rust es que no existe tal compromiso de rendimiento al elegir usar cadenas de métodos en una abstracción como un iterador en lugar de escribir un bucle imperativo manualmente. Por lo tanto, es mucho más común ver lo primero en el código.

La siguiente tabla enumera los métodos más comunes de LINQ y sus contrapartes aproximadas en Rust:

.NET	Rust	Note
`Aggregate`	`reduce`	Mira nota 1.
`Aggregate`	`fold`	Mira nota 1.
`All`	`all`	
`Any`	`any`	
`Concat`	`chain`	
`Count`	`count`	

.NET	Rust	Note
`ElementAt`	`nth`	
`GroupBy`	-	
`Last`	`last`	
`Max`	`max`	
`Max`	`max_by`	
`MaxBy`	`max_by_key`	
`Min`	`min`	
`Min`	`min_by`	
`MinBy`	`min_by_key`	
`Reverse`	`rev`	
`Select`	`map`	
`Select`	`enumerate`	
`SelectMany`	`flat_map`	
`SelectMany`	`flatten`	
`SequenceEqual`	`eq`	
`Single`	`find`	
`SingleOrDefault`	`try_find`	
`Skip`	`skip`	
`SkipWhile`	`skip_while`	
`Sum`	`sum`	
`Take`	`take`	
`TakeWhile`	`take_while`	
`ToArray`	`collect`	Mira nota 2.
`ToDictionary`	`collect`	Mira nota 2.
`ToList`	`collect`	Mira nota 2.
`Where`	`filter`	
`Zip`	`zip`	

1. La sobrecarga de `Aggregate` que no acepta un valor inicial es equivalente a `reduce`, mientras que la sobrecarga de `Aggregate` que acepta un valor inicial corresponde a `fold`.

2. `collect` en Rust generalmente funciona para cualquier tipo coleccionable, que se define como un tipo que puede inicializarse a partir de un iterador (ver `FromIterator`).

`collect` necesita un tipo de destino, que a veces el compilador tiene dificultades para inferir, por lo que el *turbofish* (`::<>`) se usa a menudo en combinación con él, como en `collect::<Vec<_>>()` . Por esta razón, `collect` aparece junto a varios métodos de extensión de LINQ que convierten una fuente enumerable/iterable en una instancia de algún tipo de colección.

El siguiente ejemplo muestra lo similar que es transformar secuencias en C# y hacer lo mismo en Rust. Primero en C#:

```
var result =
    Enumerable.Range(0, 10)
              .Where(x => x % 2 == 0)
              .SelectMany(x => Enumerable.Range(0, x))
              .Aggregate(0, (acc, x) => acc + x);

Console.WriteLine(result); // 50
```

Y en Rust:

```
let result =
    (0..10)
        .filter(|x| x % 2 == 0)
        .flat_map(|x| (0..x))
        .fold(0, |acc, x| acc + x);

println!("{result}"); // 50
```

Deferred execution (laziness)

Muchos operadores en LINQ están diseñados para ser lazy, de manera que solo realizan trabajo cuando es absolutamente necesario. Esto permite la composición o encadenamiento de varias operaciones/métodos sin causar efectos secundarios. Por ejemplo, un operador LINQ puede devolver un `IEnumerable<T>` que está inicializado, pero no produce, calcula ni materializa ningún ítem de `T` hasta que se itera sobre él. Se dice que el operador tiene *semántica de ejecución diferida*. Si cada `T` se calcula a medida que la iteración llega a él (en lugar de cuando comienza la iteración), se dice que el operador *transmite* los resultados.

Los iteradores en Rust tienen el mismo concepto de *laziness* y transmisión de resultados.

En ambos casos, esto permite representar *secuencias infinitas*, donde la secuencia subyacente es infinita, pero el desarrollador decide cómo debe terminarse la secuencia. El

siguiente ejemplo muestra esto en C#:

```
foreach (var x in InfiniteRange().Take(5))
    Console.Write($"{x} "); // Muestra "0 1 2 3 4"

IEnumerable<int> InfiniteRange()
{
    for (var i = 0; ; ++i)
        yield return i;
}
```

Rust admite el mismo concepto a través de rangos infinitos:

```
// Los generadores y yield en Rust son inestables en este momento, por lo que
// en su lugar, este ejemplo utiliza `Range`:
// https://doc.rust-lang.org/std/ops/struct.Range.html

for value in (0..).take(5) {
    print!("{value} "); // Muestra "0 1 2 3 4"
}
```

Métodos de Iterador (`yield`)

C# tiene la palabra clave `yield` que permite al desarrollador escribir rápidamente un *método de iterador*. El tipo de retorno de un método de iterador puede ser un `IEnumerable<T>` o un `IEnumerator<T>`. El compilador convierte el cuerpo del método en una implementación concreta del tipo de retorno, en lugar de que el desarrollador tenga que escribir una clase completa cada vez.

Coroutines, como se les llama en Rust, todavía se consideran una característica inestable en el momento de escribir esto.

Meta Programación

La metaprogramación puede verse como una forma de escribir código que genera o escribe otro código.

Roslyn proporciona una funcionalidad para la metaprogramación en C#, disponible desde .NET 5, llamada `Generadores de Código`. Los generadores de código pueden crear nuevos archivos fuente de C# en tiempo de compilación, que se agregan a la compilación del usuario. Antes de que se introdujeran los `Generadores de Código`, Visual Studio proporcionaba una herramienta de generación de código a través de las `Plantillas de Texto T4`. Un ejemplo de cómo funciona T4 es la siguiente plantilla o su concretización.

Rust también proporciona una funcionalidad para la metaprogramación: macros. Existen `macros declarativas` y `macros procedimentales`.

Las macros declarativas permiten escribir estructuras de control que toman una expresión, comparan el valor resultante de la expresión con patrones, y luego ejecutan el código asociado con el patrón coincidente.

El siguiente ejemplo es la definición de la macro `println!`, que es posible llamar para imprimir un texto `println!("Algún texto")`.

```
macro_rules! println {
    () => {
        $crate::print!("\n")
    };
    ($($arg:tt)*) => {{
        $crate::io::_print($crate::format_args_nl!($($arg)*));
    }};
}
```

Para aprender más sobre la escritura de macros declarativas, consulta el capítulo de la referencia de Rust sobre macros por ejemplo o El pequeño libro de macros de Rust.

Las macros procedimentales son diferentes de las macros declarativas. Estas aceptan un código como entrada, operan sobre ese código y producen un código como salida.

Otra técnica usada en C# para la metaprogramación es la reflexión. Rust no soporta reflexión.

Macros con forma de función

Las macros con forma de función tienen la siguiente forma: `function!(...)`

El siguiente fragmento de código define una macro con forma de función llamada `print_something`, que genera un método `print_it` para imprimir la cadena "Something".

En el archivo lib.rs:

```
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn print_something(_item: TokenStream) -> TokenStream {
    "fn print_it() { println!(\"Something\") }".parse().unwrap()
}
```

En el archivo main.rs:

```
use replace_crate_name_here::print_something;
print_something!();

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

Macros de tipo derive

Las macros de tipo derive pueden crear nuevos elementos dados el flujo de tokens de una estructura, enumeración o unión. Un ejemplo de una macro derive es `#[derive(Clone)]`, que genera el código necesario para que la estructura/enumeración/unión de entrada implemente el rasgo `Clone`.

Para entender cómo definir una macro derive personalizada, es posible leer la referencia de Rust sobre macros derive.

Macros de atributo

Las macros de atributo definen nuevos atributos que pueden ser adjuntados a elementos de Rust. Al trabajar con código asincrónico, si se utiliza Tokio, el primer paso será decorar el nuevo `main` asincrónico con una macro de atributo como el siguiente ejemplo:

```rust
#[tokio::main]
async fn main() {
    println!("Hola mundo");
}
```

Para entender cómo definir una macro de atributo personalizada, es posible leer la referencia de Rust sobre macros de atributo.

Programación Asíncrona

Tanto .NET como Rust admiten modelos de programación asíncronos, los cuales son similares en cuanto a su uso. El siguiente ejemplo muestra, a un nivel muy alto, cómo se ve el código asíncrono en C#:

```
async Task<string> PrintDelayed(string message, CancellationToken
cancellationToken)
{
    await Task.Delay(TimeSpan.FromSeconds(1), cancellationToken);
    return $"Message: {message}";
}
```

El código en Rust tiene una estructura similar. El siguiente ejemplo utiliza async-std para la implementación de `sleep`:

```
use std::time::Duration;
use async_std::task::sleep;

async fn format_delayed(message: &str) -> String {
    sleep(Duration::from_secs(1)).await;
    format!("Message: {}", message)
}
```

1. La palabra clave `async` en Rust transforma un bloque de código en una máquina de estados que implementa un rasgo llamado `Future`, de manera similar a como el compilador de C# transforma el código `async` en una máquina de estados. En ambos lenguajes, esto permite escribir código asíncrono de manera secuencial.

2. Cabe destacar que, tanto en Rust como en C#, los métodos/funciones asíncronos están precedidos por la palabra clave `async`, pero los tipos de retorno son diferentes. Los métodos asíncronos en C# indican el tipo de retorno completo y real porque puede variar. Por ejemplo, es común ver métodos que devuelven un `Task<T>` mientras que otros devuelven un `ValueTask<T>`. En Rust, basta con especificar el *tipo interno* `String` porque siempre será *algún futuro*; es decir, un tipo que implementa el rasgo `Future`.

3. Las palabras clave `await` están en posiciones diferentes en C# y Rust. En C#, se espera un `Task` anteponiendo la expresión con `await`. En Rust, al agregar el sufijo `.await` a la expresión se permite *encadenar métodos*, aunque `await` no sea un método.

Ver también:

- Programación asíncrona en Rust

Ejecución de tareas

En el siguiente ejemplo, el método `PrintDelayed` se ejecuta, aunque no se espere su resultado:

```
var cancellationToken = CancellationToken.None;
PrintDelayed("message", cancellationToken); // Imprime "message" después de un
segundo.
await Task.Delay(TimeSpan.FromSeconds(2), cancellationToken);

async Task PrintDelayed(string message, CancellationToken cancellationToken)
{
    await Task.Delay(TimeSpan.FromSeconds(1), cancellationToken);
    Console.WriteLine(message);
}
```

En Rust, la misma invocación no imprime nada.

```
use async_std::task::sleep;
use std::time::Duration;

#[tokio::main] // se usa para admitir un método principal asíncrono
async fn main() {
    print_delayed("message"); // No imprime nada.
    sleep(Duration::from_secs(2)).await;
}

async fn print_delayed(message: &str) {
    sleep(Duration::from_secs(1)).await;
    println!("{}", message);
}
```

Esto se debe a que los `futures` son "perezosos": no hacen nada hasta que son ejecutados. La forma más común de ejecutar un `Future` es esperarlo con `.await`. Cuando se llama `.await` en un `Future`, intentará ejecutarse hasta completarse. Si el `Future` está bloqueado, cederá el control del hilo actual. Cuando se pueda hacer más progreso, el `Future` será retomado por el ejecutor y continuará su ejecución, permitiendo que `.await` se resuelva (ver `async/.await`).

Mientras que esperar una función funciona dentro de otras funciones `async`, `main` no puede ser `async`. Esto se debe a que Rust no proporciona un entorno de ejecución para el código asíncrono. Por lo tanto, existen bibliotecas para ejecutar código asíncrono, llamadas

runtimes asíncronos. Tokio es uno de estos entornos, y se usa con frecuencia. El atributo `tokio::main` en el ejemplo anterior marca la función `async main` como el punto de entrada que será ejecutado por un entorno de ejecución, que se configura automáticamente al usar la macro.

Cancelación de tareas

Los ejemplos anteriores en C# incluían pasar un `CancellationToken` a métodos asíncronos, lo que se considera una buena práctica en .NET. Los `CancellationToken`s pueden usarse para abortar una operación asíncrona.

Dado que los `futures` en Rust son inertes (solo progresan cuando son sondeados), la cancelación funciona de manera diferente. Cuando se descarta un `Future`, este no hará más progresos. Además, descartará todos los valores instanciados hasta el punto donde el futuro esté suspendido debido a alguna operación asíncrona pendiente. Por esta razón, la mayoría de las funciones asíncronas en Rust no toman un argumento para indicar cancelación, y es por esto que a veces se refiere a descartar un futuro como *cancelación*.

`tokio_util::sync::CancellationToken` ofrece un equivalente al `CancellationToken` de .NET para señalar y reaccionar ante la cancelación, en los casos en los que implementar el rasgo `Drop` en un `Future` no sea factible.

Ejecución de múltiples tareas

En .NET, se usan frecuentemente `Task.WhenAny` y `Task.WhenAll` para manejar la ejecución de múltiples tareas.

`Task.WhenAny` se completa tan pronto como cualquiera de las tareas lo haga. Tokio, por ejemplo, proporciona la macro `tokio::select!` como alternativa a `Task.WhenAny`, lo que significa esperar en múltiples ramas concurrentes.

```
var cancellationToken = CancellationToken.None;

var result =
    await Task.WhenAny(Delay(TimeSpan.FromSeconds(2), cancellationToken),
                       Delay(TimeSpan.FromSeconds(1), cancellationToken));

Console.WriteLine(result.Result); // Esperó 1 segundo.

async Task<string> Delay(TimeSpan delay, CancellationToken cancellationToken)
{
    await Task.Delay(delay, cancellationToken);
    return $"Waited {delay.TotalSeconds} second(s).";
}
```

El mismo ejemplo en Rust:

```
use std::time::Duration;
use tokio::{select, time::sleep};

#[tokio::main]
async fn main() {
    let result = select! {
        result = delay(Duration::from_secs(2)) => result,
        result = delay(Duration::from_secs(1)) => result,
    };

    println!("{}", result); // Esperó 1 segundo.
}

async fn delay(delay: Duration) -> String {
    sleep(delay).await;
    format!("Waited {} second(s).", delay.as_secs())
}
```

Nuevamente, hay diferencias cruciales en las semánticas entre los dos ejemplos. Lo más importante es que `tokio::select!` cancelará todas las ramas restantes, mientras que `Task.WhenAny` deja al usuario la responsabilidad de cancelar cualquier tarea en curso.

De manera similar, `Task.WhenAll` puede ser reemplazado con `tokio::join!`.

Múltiples consumidores

En .NET, una `Task` puede ser usada por múltiples consumidores. Todos ellos pueden esperar la tarea y ser notificados cuando se complete o falle. En Rust, el `Future` no se puede clonar ni copiar, y al usar `await` se transfiere la propiedad. La extensión

`futures::FutureExt::shared` crea un manejador clonable de un `Future`, el cual puede distribuirse entre múltiples consumidores.

```
use futures::FutureExt;
use std::time::Duration;
use tokio::{select, time::sleep, signal};
use tokio_util::sync::CancellationToken;

#[tokio::main]
async fn main() {
    let token = CancellationToken::new();
    let child_token = token.child_token();

    let bg_operation = background_operation(child_token);

    let bg_operation_done = bg_operation.shared();
    let bg_operation_final = bg_operation_done.clone();

    select! {
        _ = bg_operation_done => {},
        _ = signal::ctrl_c() => {
            token.cancel();
        },
    }

    bg_operation_final.await;
}

async fn background_operation(cancellation_token: CancellationToken) {
    select! {
        _ = sleep(Duration::from_secs(2)) => println!("Operación en segundo
plano completada."),
        _ = cancellation_token.cancelled() => println!("Operación en segundo
plano cancelada."),
    }
}
```

Iteración asíncrona

En .NET, existen `IAsyncEnumerable<T>` y `IAsyncEnumerator<T>`, mientras que Rust aún no tiene una API para la iteración asíncrona en la biblioteca estándar. Para soportar la iteración asíncrona, el rasgo `Stream` de `futures` ofrece un conjunto de funcionalidades comparables.

En C#, escribir iteradores asíncronos tiene una sintaxis comparable a cuando se escriben iteradores sincrónicos:

```
await foreach (int item in RangeAsync(10,
3).WithCancellation(CancellationToken.None))
    Console.Write(item + " "); // Imprime "10 11 12".

async IAsyncEnumerable<int> RangeAsync(int start, int count)
{
    for (int i = 0; i < count; i++)
    {
        await Task.Delay(TimeSpan.FromSeconds(i));
        yield return start + i;
    }
}
```

En Rust, hay varios tipos que implementan el rasgo `Stream`, y por lo tanto pueden usarse para crear flujos, por ejemplo `futures::channel::mpsc`. Para obtener una sintaxis más cercana a la de C#, `async-stream` ofrece un conjunto de macros que pueden utilizarse para generar flujos de manera concisa.

```
use async_stream::stream;
use futures_core::stream::Stream;
use futures_util::{pin_mut, stream::StreamExt};
use std::{
    io::{stdout, Write},
    time::Duration,
};
use tokio::time::sleep;

#[tokio::main]
async fn main() {
    let stream = range(10, 3);
    pin_mut!(stream); // necesario para la iteración
    while let Some(result) = stream.next().await {
        print!("{} ", result); // Imprime "10 11 12".
        stdout().flush().unwrap();
    }
}

fn range(start: i32, count: i32) -> impl Stream<Item = i32> {
    stream! {
        for i in 0..count {
            sleep(Duration::from_secs(i as _)).await;
            yield start + i;
        }
    }
}
```

Estructura del Proyecto

Aunque existen convenciones sobre la estructuración de un proyecto en .NET, son menos estrictas en comparación con las convenciones de estructura de proyectos en Rust. Al crear una solución de dos proyectos usando Visual Studio 2022 (una biblioteca de clases y un proyecto de prueba xUnit), se creará la siguiente estructura:

```
.
|   BibliotecaDeClasesDeEjemplo.sln
+---BibliotecaDeClasesDeEjemplo
|       Class1.cs
|       BibliotecaDeClasesDeEjemplo.csproj
+---TestDeEjemploDelProjecto
        TestDeEjemploDelProjecto.csproj
        UnitTest1.cs
        Usings.cs
```

- Cada proyecto reside en un directorio separado, con su propio archivo `.csproj`.
- En la raíz del repositorio hay un archivo `.sln`.

Cargo utiliza las siguientes convenciones para la estructura del paquete para facilitar la inmersión en un nuevo paquete de Cargo:

```
.
+-- Cargo.lock
+-- Cargo.toml
+-- src/
|   +-- lib.rs
|   +-- main.rs
+-- benches/
|   +-- algun-bench.rs
+-- ejemplos/
|   +-- algun-ejemplo.rs
+-- tests/
    +-- algun-test-de-integracion.rs
```

- `Cargo.toml` y `Cargo.lock` se almacenan en la raíz del paquete.
- `src/lib.rs` es el archivo de biblioteca predeterminado, y `src/main.rs` es el archivo ejecutable predeterminado (ver descubrimiento automático de objetivos).
- Los benchmarks se colocan en el directorio `benches`, las pruebas de integración se colocan en el directorio `tests` (ver testing, benchmarking).
- Los ejemplos se colocan en el directorio `examples`.
- No hay un crate separado para las pruebas unitarias, las pruebas unitarias viven en el mismo archivo que el código (ver pruebas).

Gestión de proyectos grandes

Para proyectos muy grandes en Rust, Cargo ofrece workspace para organizar el proyecto. Un espacio de trabajo puede ayudar a gestionar múltiples paquetes relacionados que se desarrollan en conjunto. Algunos proyectos utilizan *manifiestos virtuales*, especialmente cuando no hay un paquete principal.

Gestión de versiones de dependencias

Al gestionar proyectos más grandes en .NET, puede ser apropiado gestionar las versiones de las dependencias de forma centralizada, utilizando estrategias como la Gestión Central de Paquetes. Cargo introdujo la herencia de workspace para gestionar las dependencias de forma centralizada.

Compilación y Building

CLI de .NET

El equivalente de la CLI de .NET (`dotnet`) en Rust es Cargo (`cargo`). Ambas herramientas son envoltorios de puntos de entrada que simplifican el uso de otras herramientas de bajo nivel. Por ejemplo, aunque podrías invocar el compilador de C# directamente (`csc`) o MSBuild a través de `dotnet msbuild` , los desarrolladores tienden a usar `dotnet build` para construir su solución. De manera similar en Rust, aunque podrías usar el compilador de Rust (`rustc`) directamente, usar `cargo build` es generalmente mucho más simple.

Building

Construir un ejecutable en .NET usando `dotnet build` restaura los paquetes, compila las fuentes del proyecto en un [ensamblado]. El ensamblado contiene el código en Lenguaje Intermedio (IL) y *típicamente* se puede ejecutar en cualquier plataforma compatible con .NET, siempre que el runtime de .NET esté instalado en el host. Los ensamblados provenientes de paquetes dependientes generalmente se ubican junto con el ensamblado de salida del proyecto. `cargo build` en Rust hace lo mismo, excepto que el compilador de Rust enlaza estáticamente (aunque existen otras opciones de enlace) todo el código en un solo binario dependiente de la plataforma.

Los desarrolladores usan `dotnet publish` para preparar un ejecutable de .NET para distribución, ya sea como un *despliegue dependiente del framework* (FDD) o un *despliegue autónomo* (SCD). En Rust, no hay un equivalente a `dotnet publish` ya que la salida de la construcción ya contiene un único binario dependiente de la plataforma para cada objetivo.

Al construir una biblioteca en .NET usando `dotnet build` , aún generará un ensamblado que contiene el IL. En Rust, la salida de la construcción es, nuevamente, una biblioteca compilada dependiente de la plataforma para cada objetivo de biblioteca.

Ver también:

- Paquetes y Crates

Dependencias

En .NET, el contenido de un archivo de proyecto define las opciones de compilación y las dependencias. En Rust, al usar Cargo, un archivo `Cargo.toml` declara las dependencias de un paquete. Un archivo de proyecto típico se verá como:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="morelinq" Version="3.3.2" />
  </ItemGroup>

</Project>
```

El equivalente de `Cargo.toml` en Rust se define como:

```
[package]
name = "hello_world"
version = "0.1.0"

[dependencies]
tokio = "1.0.0"
```

Cargo sigue una convención en la que `src/main.rs` es la raíz del crate binario con el mismo nombre que el paquete. Del mismo modo, Cargo sabe que si el directorio del paquete contiene `src/lib.rs`, el paquete contiene un crate de biblioteca con el mismo nombre que el paquete.

Paquetes

NuGet se utiliza comúnmente para instalar paquetes, y varias herramientas lo soportan. Por ejemplo, añadir una referencia a un paquete NuGet con la CLI de .NET añadirá la dependencia al archivo del proyecto:

```
dotnet add package morelinq
```

En Rust, esto funciona de manera casi igual si se usa Cargo para agregar paquetes.

```
cargo add tokio
```

El registro de paquetes más común para .NET es nuget.org, mientras que los paquetes de Rust se comparten generalmente a través de crates.io.

Análisis estático de código

Desde .NET 5, los analizadores de Roslyn vienen incluidos con el SDK de .NET y proporcionan análisis de calidad de código y estilo de código. La herramienta de linting equivalente en Rust es Clippy.

De manera similar a .NET, donde la compilación falla si hay advertencias al configurar `TreatWarningsAsErrors` en `true`, Clippy puede fallar si el compilador o Clippy emiten advertencias (`cargo clippy -- -D warnings`).

Hay otras verificaciones estáticas que considerar agregar a una pipeline de CI en Rust:

- Ejecutar `cargo doc` para asegurar que la documentación es correcta.
- Ejecutar `cargo check --locked` para asegurar que el archivo `Cargo.lock` está actualizado.