



Manejo de errores



Manejo de errores

Rust agrupa los errores en recuperables e irrecuperables.

Un error recuperable es por ejemplo un archivo no encontrado, donde tan solo se informará el error pero la ejecución del programa continuará.

Los errores irrecuperables en cambio son siempre señales de bugs en nuestro código, como por ejemplo acceder a una posición inválida de un arreglo.

En la mayoría de los lenguajes no hay distinción entre estos 2 tipos de errores y suelen manejarse con excepciones.

Rust no tiene ni maneja excepciones, en su lugar tiene el tipo `Result<T, E>` para errores recuperables y la macro `panic!` para errores irrecuperables.

Manejo de errores: panic!

```
fn main() {  
    let v = vec![1, 2, 3];  
    v[v.len()];  
}
```

```
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 3', src/main.rs:186:5  
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace  
Emanuel MacBook-Pro:src emmanuel$
```

Manejo de errores: panic!

export RUST_BACKTRACE=1

RUST_BACKTRACE=1 cargo run

```
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 3', src/main.rs:186:5
stack backtrace:
 0: rust_begin_unwind
   at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/std/src/panicking.rs:579:5
 1: core::panicking::panic_fmt
   at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/core/src/panicking.rs:64:14
 2: core::panicking::panic_bounds_check
   at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/core/src/panicking.rs:159:5
 3: <usize as core::slice::index::SliceIndex<T>>::index
   at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/core/src/slice/index.rs:260:10
 4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
   at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/core/src/slice/index.rs:18:9
 5: <alloc::vec::Vec<T,A> as core::ops::index::Index<I>>::index
   at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/alloc/src/vec/mod.rs:2703:9
 6: cripto::main
   at ./src/main.rs:186:5
 7: core::ops::function::FnOnce::call_once
   at /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/core/src/ops/function.rs:250:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Manejo de errores: call panic!

```
fn verify(data: Vec<Data>) {  
    if data.is_empty() {  
        panic!("No hay data para procesar y es obligatorio");  
    }  
    //hace mas cosas  
}
```

Manejo de errores: Result

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

Manejo de errores: Result

```
fn main() {  
    use std::io::stdin;  
    let mut buf = String::new();  
    let result = stdin().read_line(&mut buf);  
    match result {  
        Ok(i) => procesar_entrada(i),  
        Err(e) => procesar_error(e)  
    }  
}
```

Manejo de errores: Result

```
fn main() {  
    let a = "123".to_string();  
    let result = a.parse::<i32>();  
    match result {  
        Ok(data) => println!("el parseo fue correcto: {}", data),  
        Err(e) => println!("String inválido para parsear a i32: {}", a)  
    }  
}
```


Manejo de errores: Result “?”

```
fn conversion_de_tipo(s:String) -> Result<i32, ParseIntError>{  
    let dato = s.parse::<i32>()?;  
    //hago algo con dato  
    Ok(dato)  
}  
  
fn main() {  
    let result = conversion_de_tipo("1".to_string());  
    match result {  
        Ok(i) => println!("se hizo la rutina correctamente"),  
        Err(e) => println!("hubo un problema: {}", e),  
    }  
}
```

Donde más usar “?”

```
struct Persona {  
    trabajo: Option<Trabajo>,  
}  
#[derive(Clone, Copy)]  
struct Trabajo {  
    telefono: Option<NumeroTelefono>,  
}  
#[derive(Clone, Copy)]  
struct NumeroTelefono {  
    codigo_de_area: Option<u8>,  
    numero: u32,  
}
```

Donde más usar “?”

```
impl Persona {  
    fn codigo_area(&self) -> Option<u8> {  
        self.trabajo?.telefono?.codigo_de_area  
    }  
}
```

Donde más usar “?”

```
fn main() {  
    let p = Persona {  
        trabajo: Some(Trabajo {  
            telefono: Some(NumeroTelefono {  
                codigo_de_area: Some(221),  
                numero: 44444444,  
            }),  
        }),  
    };  
    println!("el codigo de area es:{:?}", p.codigo_area())  
}
```

Manejo de errores: panic o no panic

```
fn conversion_de_tipo(s:String) ->i32{  
    let dato = s.parse::<i32>().expect("Mensaje para el panic");  
    //hago algo con dato  
    return dato  
}  
  
fn main() {  
    let dato = conversion_de_tipo("1".to_string());  
    // mas instrucciones  
}
```

Manejo de errores: errores custom

```
fn main() {  
    let n = 10;  
    let r = validar_numero(n);  
    match r {  
        Ok(v) => println!("el num:{} es correcto!", v),  
        Err(e) => println!("error: {}", e),  
    }  
}
```

Manejo de errores: errores custom

```
fn validar_numero(num:i32)-> Result<i32, MiError>{  
    if num > 10{  
        return Err(MiError)  
    }  
    Ok(num)  
}
```

Manejo de errores: errores custom

```
struct MiError;  
impl Display for MiError{  
    fn fmt(&self, f: &mut std::fmt::Formatter<'>) -> std::fmt::Result {  
        write!(&f, "estoy disparando un error custom")  
    }  
}
```


Manejo de errores: errores custom con contexto

```
struct MiError(String);  
impl Display for MiError{  
    fn fmt(&self, f: &mut std::fmt::Formatter<'>) -> std::fmt::Result {  
        write!(&f, "El num: {} no es válido!", self.0)  
    }  
}
```

Manejo de errores: errores custom II

```
fn main() {  
    let u_s = Usuario{username:"staff@staff.com".to_string(),rol:Rol::Staff};  
    let u_a = Usuario{username:"admin@admin.com".to_string(),rol: Rol::Admin};  
  
    let mut p = Producto{nombre:"p1".to_string(),id:1,estado:&Estado::INI};  
  
    let r = p.cambiar_estado(&u_s);  
    match r {  
        Ok(mov) => println!("movimiento creado: {:#?}", mov),  
        Err(e) => println!("error: {}", e)  
    }  
  
    let r = p.cambiar_estado(&u_a);  
    match r {  
        Ok(mov) => println!("movimiento creado: {:#?}", mov),  
        Err(e) => println!("error: {}", e)  
    }  
}
```

Manejo de errores: errores custom II

```
#[derive(Debug)]
enum Rol{
    Admin,
    Staff,
}

#[derive(Debug)]
struct Usuario{
    username:String,
    rol:Rol
}

impl Usuario {
    fn es_staff(&self)-> bool{
        match self.rol {
            Rol::Staff => true,
            _ => false,
        }
    }

    fn to_string(&self)-> String{
        self.username.clone()
    }
}
```

Manejo de errores: errores custom II

```
struct Producto<'a>{
    nombre:String,
    estado:&'a Estado,
    id:u8,
}

impl<'a> Producto<'a>{
    fn cambiar_estado(&mut self, usuario: &'a Usuario) -> Result<Movimiento, PermisoError>{
        self.estado = self.estado.cambiar_estado(usuario)?;
        let m = Movimiento{
            usuario,
            producto_id:self.id,
            tipo:"Cambio de estado".to_string(),
        };
        Ok(m)
    }

    fn get_estado(&self) -> &Estado{
        self.estado
    }
}
```

Manejo de errores: errores custom II

```
#[derive(Debug)]  
struct Movimiento<'a>{  
    usuario:& 'a Usuario,  
    producto_id:u8,  
    tipo:String,  
}
```

Manejo de errores: errores custom II

```
enum Estado{
    INI,
    ACT,
    FIN,
}

impl Estado{
    fn estado_siguiente(&self) -> &Estado{
        match self{
            Estado::INI=> &Estado::ACT,
            Estado::ACT=> &Estado::FIN,
            Estado::FIN=> &Estado::FIN,
        }
    }

    fn es_ini(&self) -> bool{
        match self {
            Estado::INI => true,
            _ => false
        }
    }
}
```

Manejo de errores: errores custom II

```
fn to_string(&self) -> String{
    match self{
        Estado::INI => "INI".to_string(),
        Estado::ACT => "ACT".to_string(),
        Estado::FIN => "FIN".to_string(),
    }
}

fn cambiar_estado<'a>(&self, usuario:&'a Usuario)-> Result<&Estado, PermisoError>{
    let result = Permisos::puede_cambiar_estado(&self, self.estado_siguiente(), usuario);
    match result {
        Ok(v) => Ok(self.estado_siguiente()),
        Err(e) => Err(e)
    }
}
}
```

Manejo de errores: errores custom II

```
#[derive(Debug)]
struct Permisos;
impl Permisos{
    fn puede_cambiar_estado<'a>(
        estado_actual:&'a Estado,
        estado_siguiete:&Estado,
        usuario:&Usuario)-> Result<&'a Estado, PermisoError>{
        if usuario.es_staff() && estado_actual.es_ini(){
            return Err(PermisoError(
                usuario.to_string(),
                estado_siguiete.to_string()) )
        }
        Ok(estado_actual)
    }
}
```


Manejo de errores: errores custom II

```
use std::fmt::Display;

#[derive(Debug, Clone)]

struct PermisoError(String, String);

impl Display for PermisoError {

    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {

        write!(f, "el usuario: {} no posee los permisos para realizar de cambio de estado a: {}", self.0,

self.1)

    }

}
```