Luca Rinaldi
Sviluppatore Develer

Rust: il filo rosso che unisce Option, Result e Future

Techlabs

17/05/2023

develer

INTRODUZIONE

L'obiettivo è dare contesto ad alcuni costrutti base di Rust e come poterli sfruttare al meglio.

Sommario

- Introdurremo il concetto di Monade
- Gestione del null: Option
- Gestione degli errori: Result
- Rust asincrono e Futures
- Esempio pratico
- Conclusioni e suggerimenti

Introduzione alle Monadi

Sono spesso utilizzate in linguaggi funzionali, e.g. Haskell.

Sono utilizzate per esprimere computazioni impure con IO e gestione degli errori.

Monadi: in teoria

Le Monadi derivano dalla Teoria delle Categoria.

Ha le seguenti caratteristiche

- un tipo `M T`
- un costruttore `unit : T → M T`
- funzione `fmap` `fmap : (M T, T → U) → M U`
- funzione `bind` `bind : (M T, T → M U) → M U`

Funtori

Monadi

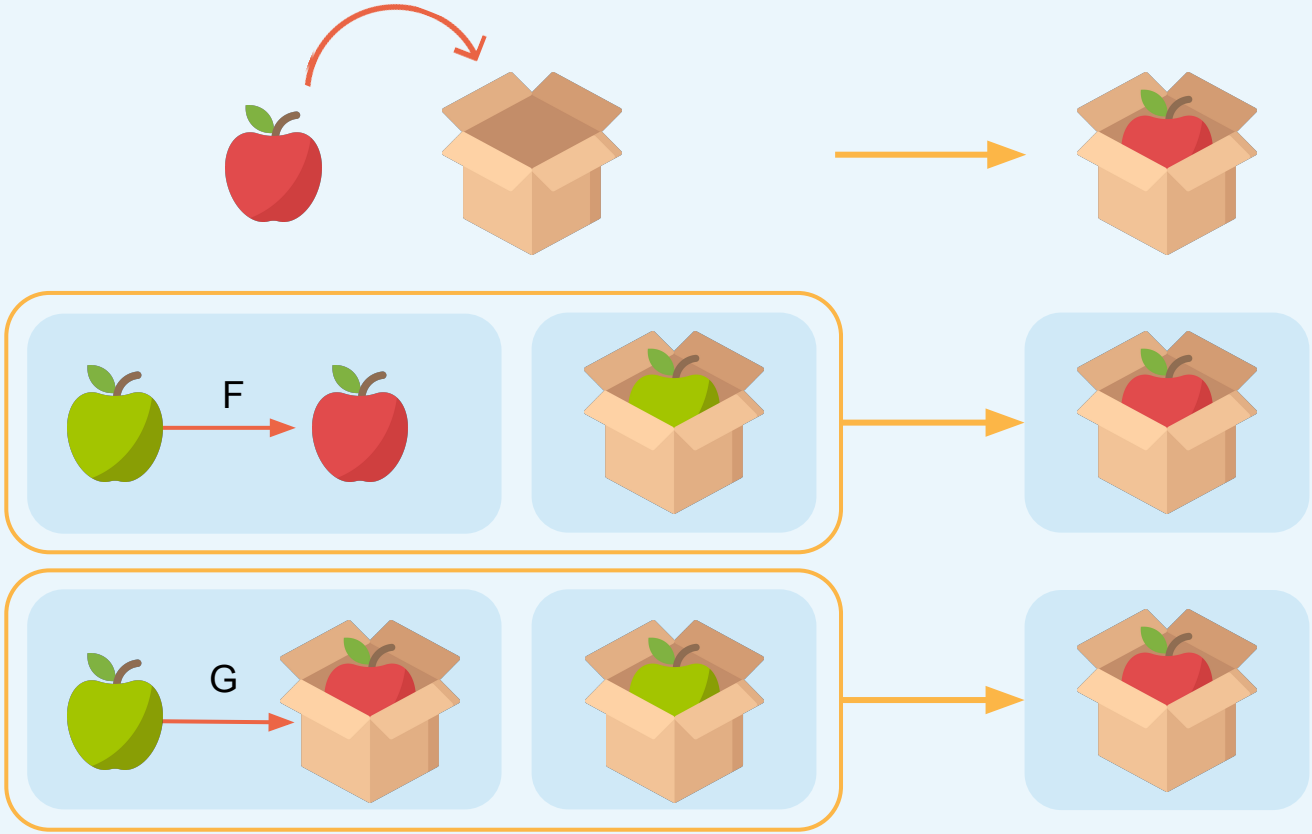
Le Monadi sono `Funtori` con una operazione in più detta `bind`.

Monadi con le mele

Costruire una Monade

Usare `fmap` per applicare F

Usare `bind` per applicare G



Rust e i linguaggi funzionali

Rust ha preso a piene mani dai linguaggi funzionali e concetti di informatica teorica

L'ownership rule è un esempio su tutti.

Rust e le Monadi

Molti dei costrutti base rispettano le proprietà delle:

- Option
- Result
- Future
- Iterator

La gestione del NULL: Option

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965.*
>
> – Tony Hoare

Gestione del NULL: Option

In Rust non esiste il valore **NULL**. Per codificare l'assenza di un valore si usa `Option`.

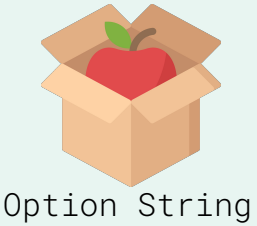
```
enum Option<T> {
   None,
   Some(T),
}

let x: Option<i32> = Some(7);
let y: Option<i32> = None;
```

La differenza principale è che di default un valore non può essere mai **NULL**

Option è una Monade

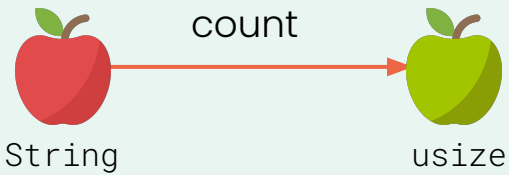
Il costruttore

```rust
let opt: Option<String> = Some(String::from("foo"));
```



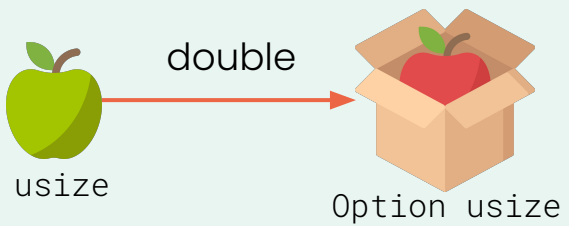
L'operazione `fmap`

```rust
let count = |x: String| x.len();
let opt: Option<usize> = opt.map(count);
assert_eq!(opt, Some(3));
```



L'operazione `bind`

```rust
let double = |x: usize| x.checked_mul(2);
let opt: Option<usize> = opt.and_then(double);
assert_eq!(opt, Some(6));
```



La gestione degli errori: Result

Gestione degli errori: Result

Rust segue la filosofia degli errori come valori. Quindi invece delle eccezioni si usa `Result`.

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

let x: Result<i32, String> = Ok(7);
let y: Result<i32, String> = Err(String::from("invalid value"));
```

Questo approccio permette di avere un unico percorso per il ritorno di errori o valori validi.

Result è una Monade

Il costruttore

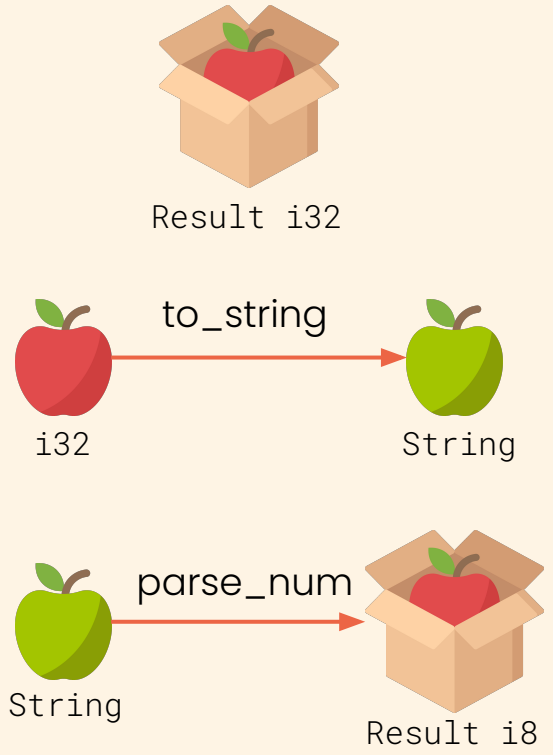
```
type Error = ParseIntError;
let res: Result<i32, Error> = Ok(17);
```

L'operazione `fmap`

```
let to_string = |x: i32| x.to_string();
let res: Result<String, _> = res.map(to_string);
assert_eq!(res, Ok(String::from("17")));
```

L'operazione `bind`

```
let parse_num = |x: String| i8::from_str(&x);
let res: Result<i8, _> = res.and_then(parse_num);
assert_eq!(res, Ok(17_i8));
```



Definire un errore in Rust

Rust incoraggia la definizione di errori specifici per la propria applicazione/libreria.

Gli errori devono implementare il trait `std::error::Error`.

```
pub trait Error: Debug + Display {
    fn source(&self) -> Option<&(dyn Error + 'static)>;
}
```

Il messaggio d'errore deve essere "printabile" attraverso il trait `fmt::Display` e non deve contenere punteggiatura o maiuscole.

Convertire gli errori

Result mette a disposizione la funzione `.map_err` per convertire un errore da un tipo ad un altro.

```
struct MyError1;
struct MyError2;

let res: Result<i32, MyError1> = Err(MyError1);
let res: Result<i32, MyError2> = res.map_err(|_e| MyError2);
assert_eq!(res, Err(MyError2));
```

I vantaggi delle monadi

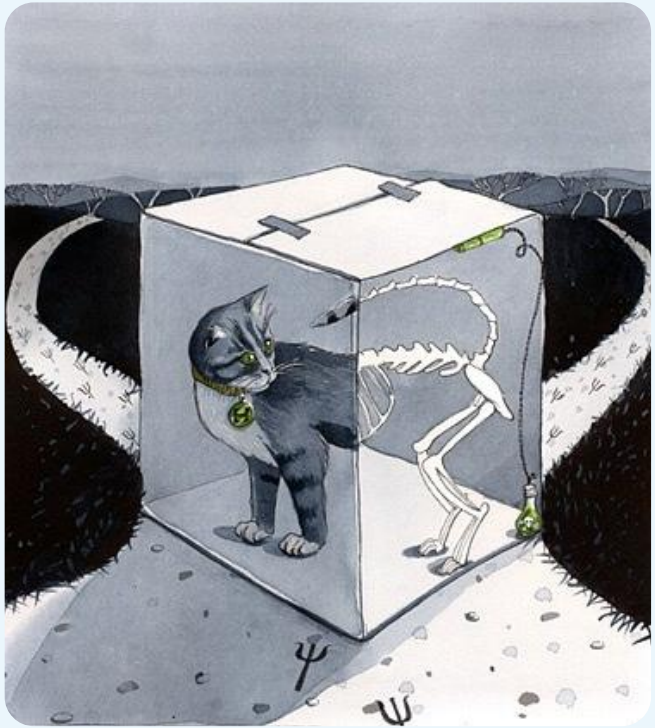
Monadi come gatto di Schrödinger

Le operazioni `.map` e `.and_then` permettono di manipolare il contenuto della “scatola” a prescindere che sia:

- un valore valido, oppure
- un errore/valore nullo

Per esempio il seguente codice sarà valido

```
let maybe_null: Option<i32> = None;
let maybe_null = maybe_null.map(|x| x + 1);
assert_eq!(maybe_null, None);
```



“Do notation”: Il ? in Rust

L’operazione `bind` (`.and_then`) è un modo per esprimere una computazione sequenziale in un linguaggio funzionale.

Rust mette a disposizione uno zucchero sintattico per convertire questa sequenza in sintassi imperativa.

```
let opt: Option<i32> = Some(7);
let opt = opt
    .and_then(|x: i32| action1(x))
    .and_then(|x: i32| action2(x))
    .and_then(|x: i32| action3(x));
```

```
fn foo() -> Option<i32> {
    let opt: Option<i32> = Some(7);
    let x: i32 = opt?;
    let x: i32 = action1(x)?;
    let x: i32 = action2(x)?;
    action3(x)
}
```

Funzioni utili di Option e Result

.ok(), convertire Result in Option

```
let res: Result<i32, &str> = Ok(5);
let opt: Option<i32> = res.ok();
```

.ok_or(), convertire Option in Result

```
let opt: Option<i32> = Some(5);
let opt: Result<i32, &str> = opt.ok_or("error");
```

.collect(), create un Result/Option da un iteratore

```
let results = [Ok(5_i32), Ok(6_i32), Err("foo"), Ok(7_i32)];
let lst: Result<Vec<i32>, &str> = results.iter().cloned().collect();
assert_eq!(lst, Err("foo"));
```

Result/Option: funzioni utili (I)

`.ok()`, convertire Result in Option

```
let res: Result<i32, &str> = Ok(5);
let opt: Option<i32> = res.ok();
```

`.ok_or()`, convertire Option in Result

```
let opt: Option<i32> = Some(5);
let opt: Result<i32, &str> = opt.ok_or("error");
```

`.traspose()`, invertire Result/Option

```
let x: Option<Result<i32, &str>> = Some(Ok(5));
let y: Result<Option<i32>, &str> = Ok(Some(5));
assert_eq!(x.transpose(), y);
```

Result/Option: funzioni utili (II)

`.collect()`, create un Results/Options di un vettore

```
let results = [Ok(5_i32), Ok(6_i32), Err("foo"), Ok(7_i32)];
let lst: Result<Vec<i32>, &str> = results.iter().cloned().collect();
assert_eq!(lst, Err("foo"));
```

`.partition_result()`, create un vettore di Ok e un vettore di Err

```
use itertools::Itertools;
let (successes, failures): (Vec<i32>, Vec<&str>) = results.iter().cloned().partition_result();
assert_eq!(successes, [5_i32, 6_i32, 7_i32]);
assert_eq!(failures, ["foo"]);
```

Rust asincrono

Future

Rust permette di definire computazioni asincrone attraverso le Future.
Le Future sono tipi che implementano il trait `std::future::Future`.

```rust
pub trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

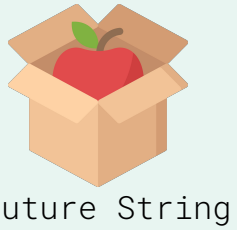
```rust
let fut /*: impl Future<Output = i32> */ = async { 1_i32 };

let x: i32 = fut.await;
assert_eq!(x, 1);
```

- `async`, crea un blocco/funzioni asincrone
- `await`, attende una future

Future è una monade

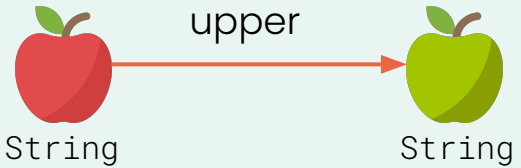
Il costruttore

```
let fut =  async { String::from("Luca") };
```



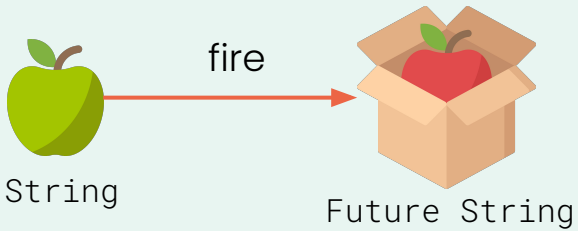
L'operazione `fmap`

```
let upper = |x: String| x.to_uppercase();
let fut = fut.map(upper);
```



L'operazione `bind`

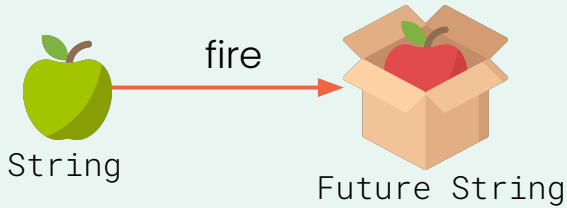
```
let fire = |x: String| async move { format!("🔥{x}🔥")};
let fut = fut.then(fire);
```



await è la “do notation” delle future

`await` è uno zucchero sintattico per trasformare la composizione di funzioni asincrone con `.then` in codice imperativo.

```
let fire = |x: String| async move { format!("🔥{x}🔥")};
```




```
let fut = fut.then(fire);
```

```
let fut = async {
    let x: String = fut.await;
    fire(x).await
};
```

Esempio sulla gestione degli errori

Esempio: Gestione degli errori (I)

@lucarin91/monad-talk-example

Piccolo programma che carichi una struttura `Person` da un file.

Gli obiettivi dell'esempio sono:

- vedere come gestire gli errori in rust in modo idiomatico
- confrontare l'utilizzo di `.and_then`/`.then` con `?`/`.await`
- infine useremo la libreria `thiserror` per generare parte del codice

```
// person.txt
luca,rinaldi
```

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

Esempio: Gestione degli errori (II)

```rust
impl FromStr for Person {
    type Err = PersonError;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        let mut it = s.split(',').map(str::to_owned);

        let name = it.next().ok_or(PersonError::Parse)?;
        let surname = it.next().ok_or(PersonError::Parse)?;

        if it.next().is_some() {
            return Err(PersonError::Parse);
        }

        Ok(Person { name, surname })
    }
}
```

```rust
// main.rs

let txt: &str = "luca,rinaldi";

let p = Person::from_str(txt).unwrap();
let p: Person = txt.parse().unwrap();
```

Esempio: Gestione degli errori (III)

```rust
#[derive(Debug)]
enum PersonError {
  Io(io::Error),
  Utf8(FromUtf8Error),
  Parse,
}
```

```rust
impl From<io::Error> for PersonError {
  fn from(err: io::Error) -> PersonError {
      PersonError::Io(err)
  }
}
impl From<FromUtf8Error> for PersonError {
  fn from(err: FromUtf8Error) -> PersonError {
      PersonError::Utf8(err)
  }
}
```

```rust
impl fmt::Display for PersonError {
   fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Resul
       match self {
           PersonError::Io(e) => write!(f, "cannot read file: {e}"),
           PersonError::Utf8(e) => write!(f, "not a valid string: {e}"),
           PersonError::Parse => write!(f, "invalid person format"),
       }
   }
}
```

Esempio: Gestione degli errori (IV)

```rust
fn read_person(path: &str) -> impl Future<Output = Result<Person, PersonError>> + '_ {
    tokio::fs::read(path).map(|r: Result<Vec<u8>, io::Error>| {
        r.map_err(PersonError::from)
            .and_then(|b: Vec<u8>| String::from_utf8(b).map_err(PersonError::from))
            .and_then(|s: String| Person::from_str(&s))
    })
}
```

```rust
async fn read_person(path: &str) -> Result<Person, PersonError> {
    let b: Vec<u8> = tokio::fs::read(path).await?;
    let s: String = String::from_utf8(b)?;
    let p: Person = Person::from_str(&s)?;
    Ok(p)
}
```

Esempio: Gestione degli errori (V)

Rimuoviamo molto del codice ripetitivo con la crate `thiserror`.

```rust
#[derive(thiserror::Error, Debug)]
enum PersonError {
  #[error("cannot read file: {0}")]
  Io(#[from] io::Error),

  #[error("not a valid string: {0}")]
  Utf8(#[from] FromUtf8Error),

  #[error("invalid person format")]
  ParseError,
}
```

Conclusioni e suggerimenti

Take home concepts

Option e Result

- `.map`, applicare una trasformazione non fallibile
- `.and_then`, applicare una trasformazione fallibile
- una sequenza di `.and_then` può essere convertita in un blocco di `?`

Future

- `.then` combina più computazione asincrone
- `await` ha lo stesso ruolo di `?` in Option/Result
- una sequenza di `.then` può essere convertita in `await`

Fonti

- *Category Theory for Programmers: The Preface, Bartosz Milewski, 2014*
- *What is a Monad? - Computerphile*
- *What is a monad? And who needs Haskell anyway?, users.rust-lang.org, 2020*
- *Beginner's guide to Error Handling in Rust, sheshbabu.com, 2020*

Q&A

Torna RustLab! 🦀

Dal **19 al 21 novembre 2023** al Grand Hotel Mediterraneo (FI) torna la conferenza internazionale su Rust organizzata da Develer.

Novità! Quest'anno RustLab e GoLab saranno contemporanee e nella stessa location! Due conferenze al prezzo di una!

Questo è un coupon di sconto del 10% che puoi utilizzare anche sulla tariffa Early Bird: **DTL10W**

Luca Rinaldi

lucarin@develer.com

Vuoi rimanere aggiornato sugli eventi Develer?
Seguici nei nostri canali social:

develer

www.develer.com