Fermetures, Itérateurs

Frédéric Wagner

2024

Ensimag

Plan

Fermetures

Itérateurs

Fonctions en paramètres

- il arrive souvent que l'on souhaite passer une fonction en paramètre d'une autre fonction

Fonctions en paramètres

- il arrive souvent que l'on souhaite passer une fonction en paramètre d'une autre fonction
- par exemple pour créer un thread en C on utilise

```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg);
```

  - `start_routine` est un pointeur de fonction

Fonctions en paramètres

- il arrive souvent que l'on souhaite passer une fonction en paramètre d'une autre fonction
- par exemple pour créer un thread en C on utilise

```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg);
```

  - `start_routine` est un pointeur de fonction
- on peut faire ça en **Rust** mais un pointeur est une indirection donc un surcoût

Fonctions en arguments

- solution naturelle : utiliser des types génériques et des traits

Fonctions en arguments

- solution naturelle : utiliser des types génériques et des traits
- exemple : la méthode **map** de l'**Option**

```
pub fn map<U, F>(self, f: F) -> Option<U>
where
  F: FnOnce(T) -> U {
      match self {
          None => None,
          Some(c) => Some(f(c)),
      }
}
```

Fn, FnMut, FnOnce

- hiérarchie de traits qui caractérisent les fonctions

Fn, FnMut, FnOnce

- hiérarchie de traits qui caractérisent les fonctions
  - **FnOnce**
    - les fonctions qu'on peut appeller mais peut-être une seule fois
  - **FnMut** (requiert **FnOnce**) les fonctions qu'on peut appeller plusieurs fois mais qui peut-être changent par effet de bord l'état de variables externes
  - **Fn** (requiert **FnMut**) les fonctions qu'on peut appeller plusieurs fois et n'utilisent pas d'accès en écriture à des variables externes

Fn, FnMut, FnOnce

- hiérarchie de traits qui caractérisent les fonctions
  - **FnOnce**
    - les fonctions qu'on peut appeller mais peut-être une seule fois
  - **FnMut** (requiert **FnOnce**) les fonctions qu'on peut appeller plusieurs fois mais qui peut-être changent par effet de bord l'état de variables externes
  - **Fn** (requiert **FnMut**) les fonctions qu'on peut appeller plusieurs fois et n'utilisent pas d'accès en écriture à des variables externes
- dans l'exemple du **map**

```
F: FnOnce(T) -> U
```

  - **F** est une fonction qu'on ne peut appeller éventuellement qu'une seule fois, prend un **T** et renvoie un U

Utilisation basique

```
let o = Some(3);
let o2 = None;
fn increment(x: u32) -> u32 { x + 1 }
fn to_i32(x: u32) -> i32 { x as i32 }
let r = o.map(increment);
let r2 = o2.map(to_i32);
assert_eq!(r, Some(4));
assert_eq!(r2, None);
```

Utilisation basique

```
let o = Some(3);
let o2 = None;
fn increment(x: u32) -> u32 { x + 1 }
fn to_i32(x: u32) -> i32 { x as i32 }
let r = o.map(increment);
let r2 = o2.map(to_i32);
assert_eq!(r, Some(4));
assert_eq!(r2, None);
```

- note : on peut voir un code sans conditionnelles

Fermeture

- besoin d'une manière rapide de déclarer des fontions anonymes (lambda en python)
- fermeture : `|argument1, argument2, ...| resultat`

```
let r = Some(3).map(|e| e+1);
```

Fermeture

- besoin d'une manière rapide de déclarer des fontions anonymes (lambda en python)
- fermeture : `|argument1, argument2, ...| resultat`

```
let r = Some(3).map(|e| e+1);
```

- une fermeture peut emprunter des variables de son environnement

```
let mut entiers_traites = Vec::new();
let increment = |e| {
    entiers_traites.push(e);
    e + 1
};
let mut o = Some(3);
o = o.map(increment);
o = o.map(increment);
assert_eq!(entiers_traites, vec![3,4]);
```

  - ici elle emprunte le vecteur mutablement

Compilation d'une fermeture

- en assembleur, qu'est-ce qu'une fermeture ?

```
let mut y = 0;
let x = 3;
let f = |e| { y += 1; e + x + y };
f(0);
```

Compilation d'une fermeture

- en assembleur, qu'est-ce qu'une fermeture ?

```
let mut y = 0;
let x = 3;
let f = |e| { y += 1; e + x + y };
f(0);
```

- structure de donnée générée par le compilateur
  - contient toutes les références requises
  - ici **f** est sur la pile
  - implémente **FnMut**
  - l'appel **f** appelle la méthode **call_mut** du trait

Fn, FnMut, FnOnce

```rust
let x = 3;
test_fn(|e| e + x);
let mut y = 0;
test_fnmut(|e| {
    y += 1;
    e + y
});
let mut v = Vec::new();
test_fnonce(move |e| {
    v.push(e);
    v
});
```

```rust
fn test_fn<F>(f: F)
where F: Fn(u32) -> u32 {
    println!("{}", f(0));
}
fn test_fnmut<F>(mut f: F)
where F: FnMut(u32) -> u32 {
    println!("{}", f(0));
}
fn test_fnonce<F>(f: F)
where F: FnOnce(u32) -> Vec<u32> {
    println!("{:?}", f(0));
}
```

D'autres exemples d'utilisation

```
let r = if a > b { Some(a/b) } else { Some(b/a) };
let r = (a>b).then(|| a/b).or_else(|| b/a);
```

- juste sur **Option**
  - `and_then`
  - `filter`
  - `unwrap_or_else`
  - `inspect`
  - ...

Plan

Fermetures

Itérateurs

For

```
for e in 0..10 { ... }
for x in [1,2,3] { ... }
```

- **for** sur de nombreux types différents

For

```
for e in 0..10 { ... }
for x in [1,2,3] { ... }
```

- **for** sur de nombreux types différents
- interface commune : **IntoIterator**
- itérable : type sur lequel on peut boucler
  - l'itérable peut se transformer en itérateur

```
pub trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item = Self::Item>;

    fn into_iter(self) -> Self::IntoIter;
}
```

Exemple

```rust
impl<T, A> IntoIterator for Vec<T, A>
where
    A: Allocator,
```

```rust
fn into_iter(self) -> <Vec<T, A> as IntoIterator>::IntoIter
```

```rust
type Item = T
```

The type of the elements being iterated over.

```rust
type IntoIter = IntoIter<T, A>
```

Which kind of iterator are we turning this into?

- transfert de propriété du vec à l'itérateur

Itérateur

- itérateur : stocke l'état interne des itérations
  - par exemple, si on boucle sur un tableau, il faut savoir quel est l'indice de l'élément courant
  - méthode **next** : renvoie l'élément suivant et met à jour l'état interne

std::iter

Trait Iterator

Since 1.0.0 · Source

```
pub trait Iterator {
    type Item;

    // Required method
    fn next(&mut self) -> Option<Self::Item>;
```

Le sucre

```
for x in i {
    println!("x: {x}");
}
```

se désucre en :

```
let mut iterateur = i.into_iter();
while let Some(x) = iterateur.next() {
    println!("x: {x}");
}
```

Écriture d'un itérateur

```
struct Puissances {
    base: u32,
    courant: u32,
    n: u32,
}
impl Iterator for Puissances {
    type Item = u32;
    fn next(&mut self) -> Option<Self::Item> {
        (self.n != 0).then(|| {
            self.n -= 1;
            self.courant *= self.base;
            self.courant
        })
    }
}
```

Utilisation

```
fn puissances(base: u32, n: u32) -> Puissances {
    Puissances {
        base,
        courant: 1,
        n,
    }
}

for p in puissances(2, 3) {
    println!("{p}"); // 2, 4, 8
}
```

Utilisation

```
fn puissances(base: u32, n: u32) -> Puissances {
    Puissances {
        base,
        courant: 1,
        n,
    }
}

for p in puissances(2, 3) {
    println!("{p}"); // 2, 4, 8
}
```

- note : tous les itérateurs sont aussi des itérables

Méthodes

- 76 méthodes sur le trait de base
- des traits additionnels
  - **Itertools** dans la crate itertools
  - **DoubleEndedIterator**
  - ...

Méthodes

- 76 méthodes sur le trait de base
- des traits additionnels
  - **Itertools** dans la crate itertools
  - **DoubleEndedIterator**
  - ...
- il faut pas mal de temps pour être à l'aise avec tout le monde

Utilité

- séparer la manière de boucler du traitement
  - réutilisation
  - simplification du code
- + de généricité
- - de consommation mémoire

Utilité

- séparer la manière de boucler du traitement
  - réutilisation
  - simplification du code
- + de généricité
- - de consommation mémoire
- exemple : min et max

Quelques classiques

- sum

```
assert_eq!([1,3,2].iter().sum::<u32>(), 6);
```

- product

```
fn factorielle(n: u64) -> u64 {
    (2..n).product()
}
```

Quelques classiques

- sum

```
assert_eq!([1,3,2].iter().sum::<u32>(), 6);
```

- product

```
fn factorielle(n: u64) -> u64 {
    (2..n).product()
}
```

- note : l'inférence de type a du mal avec sum et product

Quelques classiques

- map
  - applique une fonction sur chaque élément

```
fn map<B, F>(self, f: F) -> Map<Self, F>
where
    Self: Sized,
    F: FnMut(Self::Item) -> B,
```

- filter
  - boucle sur les éléments qui valident un prédicat

```
fn filter<P>(self, predicate: P) -> Filter<Self, P>
where
    Self: Sized,
    P: FnMut(&Self::Item) -> bool,
```

Quelques classiques

- enumerate
  - boucle sur des couples (numéro d'itération, élément)

```
let tab = [4, 2, 7, 3, 1, 5];
let indices_impairs = tab.iter()
    .enumerate()
    .filter_map(|(i, e)| (e & 1 != 0).then_some(i));
let premier_impair = indices_impairs.next();
assert_eq!(premier_impair, Some(3));
```

Quelques classiques

- zip
  - boucle sur deux itérateurs
  - s'arrète au plus court
- exemple : produit scalaire

Quelques classiques

- zip
  - boucle sur deux itérateurs
  - s'arrète au plus court
- exemple : produit scalaire

```
fn produit_scalaire(a: &[u32], b: &[u32]) -> u32 {
    a.iter().zip(b).map(|(ea, eb)| *ea * *eb).sum()
}
```

Quelques classiques

- chain
  - chaine deux itérateurs
  - souvent sans surcoûts
- cycle
  - répète un itérateur à l'infini

Quelques classiques

- chain
  - chaine deux itérateurs
  - souvent sans surcoûts
- cycle
  - répète un itérateur à l'infini

```
let couleurs = ["red", "green"];
let mots = ["foo", "bar", "baz"];
for (i, (couleur, mot)) in couleurs.iter()
      .cycle()
      .zip(&mots)
      .enumerate() {
  println!("<text x='0' y='{}' stroke='{couleur}'>
            {mot}</text>", i*10)
}
```

Quelques classiques

- nth
  - renvoie le $n^{ème}$ élément
- find
  - renvoie le premier élément vérifiant un prédicat
- skip
  - saute les premiers éléments
- take
  - saute les derniers éléments

Quelques classiques

```
fn flat_map<U, F>(self, f: F) -> FlatMap<Self, U, F> ⓘ
where
    Self: Sized,
    U: IntoIterator,
    F: FnMut(Self::Item) -> U,
```

Creates an iterator that works like map, but flattens nested structure.

- boucle sur le contenu des itérables générés par le **map**
- génère parfois des problèmes délicats d'emprunts

Quelques classiques

```rust
fn flat_map<U, F>(self, f: F) -> FlatMap<Self, U, F> ⓘ
where
    Self: Sized,
    U: IntoIterator,
    F: FnMut(Self::Item) -> U,
```

Creates an iterator that works like map, but flattens nested structure.

- boucle sur le contenu des itérables générés par le **map**
- génère parfois des problèmes délicats d'emprunts

```rust
let combinaisons = a
  .iter()
  .enumerate()
  .flat_map(|(i, e1)| a[..i].iter().map(move |e2| (e1, e2)));
```

Tranches, chaines et itérateurs

- les tranches (rappel : `&[T]` ou `&mut[T]`) fournissent différents itérateurs utiles
  - **chunks, chunksmut** : boucle par blocs
  - **windows** : fenêtre **glissante**
- les chaines (`&str`) également
  - **chars** : boucle sur les caractères
  - **lines** : sur les lignes
  - **split_whitespace** : sur les mots
  - **matches** : sur des motifs