Traits et types génériques

Frédéric Wagner

2024

Ensimag

Plan

Traits

Traits classiques

Traits génériques et types associés

Recherche du max

```
fn recherche_max<T>(tab: Vec<T>) -> Option<T> {
    let mut max = None;
    for e in tab {
        if Some(e) > max {
            max = Some(e)
        }
    }
    max
}
```

Recherche du max

```
fn recherche_max<T>(tab: Vec<T>) -> Option<T> {
    let mut max = None;
    for e in tab {
        if Some(e) > max {
            max = Some(e)
        }
    }
    max
}
```

- ne compile pas, pourquoi ?

Diagnostic

```
error[E0369]: binary operation `>` cannot be applied to type `Option<T>`
 --> src/main.rs:4:20
  |
4 |         if Some(e) > max {
  |            ------- ^ --- Option<T>
  |            |
  |            Option<T>
  |
help: consider restricting type parameter `T`
  |
1 | fn recherche_max<T: std::cmp::PartialOrd>(tab: Vec<T>) -> Option<T> {
  |                   ++++++++++++++++++++++
```

- il faut un ordre partiel sur `T`

PartialOrd

std::cmp

Trait PartialOrd

Since 1.0.0 · Source

Settings

```
pub trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where
    Rhs: ?Sized,
{
    // Required method
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;

    // Provided methods
    fn lt(&self, other: &Rhs) -> bool { ... }

    fn le(&self, other: &Rhs) -> bool { ... }

    fn gt(&self, other: &Rhs) -> bool { ... }

    fn ge(&self, other: &Rhs) -> bool { ... }
}
```

Traits

- un trait représente une capacité d'un type
  - souvent de fournir une méthode mais pas seulement

Traits

- un trait représente une capacité d'un type
  - souvent de fournir une méthode mais pas seulement
- les fonctions génériques peuvent requérir des traits sur leurs types génériques

Traits

- un trait représente une capacité d'un type
  - souvent de fournir une méthode mais pas seulement
- les fonctions génériques peuvent requérir des traits sur leurs types génériques
  - dans notre exemple

```
fn recherche_max<T>(tab: Vec<T>) -> Option<T> where
    T: std::cmp::PartialOrd
```

Traits

- un trait représente une capacité d'un type
  - souvent de fournir une méthode mais pas seulement
- les fonctions génériques peuvent requérir des traits sur leurs types génériques
  - dans notre exemple

```
fn recherche_max<T>(tab: Vec<T>) -> Option<T> where
    T: std::cmp::PartialOrd
```

- PartialOrd fournit
  - une méthode `fn gt(&self, other: &Rhs)->bool` (utilisée implicitement)
  - une **garantie conceptuelle** que la fonction de comparaison respecte un ordre partiel

Déclarer un trait

```
trait Point: Sized {
    fn x(&self) -> f64;
    fn y(&self) -> f64;
    fn new(x: f64, y: f64) -> Self;
    fn milieu(&self, p: &Self) -> Self {
        Self::new((self.x() + p.x()) / 2., (self.y() + p.y()))
    }
}
```

- liste de méthodes du trait
  - certaines requises
  - certaines optionnelles
    - spécialisables
- un trait peut pré-requérir d'autres traits
  - par exemple **Ord** requiert **PartialOrd**
  - ici, **Point** requiert **Sized**

Implémenter un trait

- implémenter au moins les méthodes requises

```
impl Point for (f64, f64) {
    fn x(&self) -> f64 { self.0 }
    fn y(&self) -> f64 { self.1 }
    fn new(x: f64, y: f64) -> Self {
        (x, y)
    }
}
```

- certains traits sont dérivables
- certains traits sont auto-implémentés

Utilisation des traits

```
// NOTE: le trait Point doit être visible ici
let p1 = (2., 6.);
let p2 = (4., 8.);
assert_eq!(p1.milieu(&p2), (3., 7.));
```

- permet d'obtenir des super pouvoirs juste en implémentant les pré-requis
- permet de clarifier le contrat des fonctions
- permet d'enrichir des types existants avec de nouvelles méthodes
- le système de typage de rust est turing-complet

Orphan Rule

- tous les traits ne sont pas implémentables par tout le monde

Orphan Rule

- tous les traits ne sont pas implémentables par tout le monde
- une seule implémentation de chaque trait par type
  - y-compris dans le futur

```
impl<P: Pomme> Fruit for P {...}
impl<P: Poire> Fruit for P {...}
```

Orphan Rule

- tous les traits ne sont pas implémentables par tout le monde
- une seule implémentation de chaque trait par type
  - y-compris dans le futur

```
impl<P: Pomme> Fruit for P {...}
impl<P: Poire> Fruit for P {...}
```

- quid des bibliothèques ?

Orphan Rule

- tous les traits ne sont pas implémentables par tout le monde
- une seule implémentation de chaque trait par type
  - y-compris dans le futur

```
impl<P: Pomme> Fruit for P {...}
impl<P: Poire> Fruit for P {...}
```

- quid des bibliothèques ?
- orphan rule : pour implémenter un trait sur un type il faut :
  - soit avoir créé le trait
  - soit avoir créé le type

Orphan Rule

- tous les traits ne sont pas implémentables par tout le monde
- une seule implémentation de chaque trait par type
  - y-compris dans le futur

```
impl<P: Pomme> Fruit for P {...}
impl<P: Poire> Fruit for P {...}
```

- quid des bibliothèques ?
- orphan rule : pour implémenter un trait sur un type il faut :
  - soit avoir créé le trait
  - soit avoir créé le type
- il existe des solutions de contournement

Plan

Traits

Traits classiques

Traits génériques et types associés

Display and Debug

- deux traits pour afficher des variables en texte
- **Debug** est dérivable

```
#[derive(Debug)]
struct Foo<T> {
    bar: u32,
    baz: T
}
let f = Foo {bar: 3, baz: 4u32};
println!("f est {f:?}");
```

```
f est Foo { bar: 3, baz: 4 }
```

Default

- nouvelle variable avec une valeur par défaut

```
pub trait Default: Sized {
    fn default() -> Self;
}
```

- implémenté par de nombreux types

```
let v: Vec<u32> = Default::default();
assert_eq!(v, vec![]);
let e: u32 = Default::default();
assert_eq!(e, 0);
```

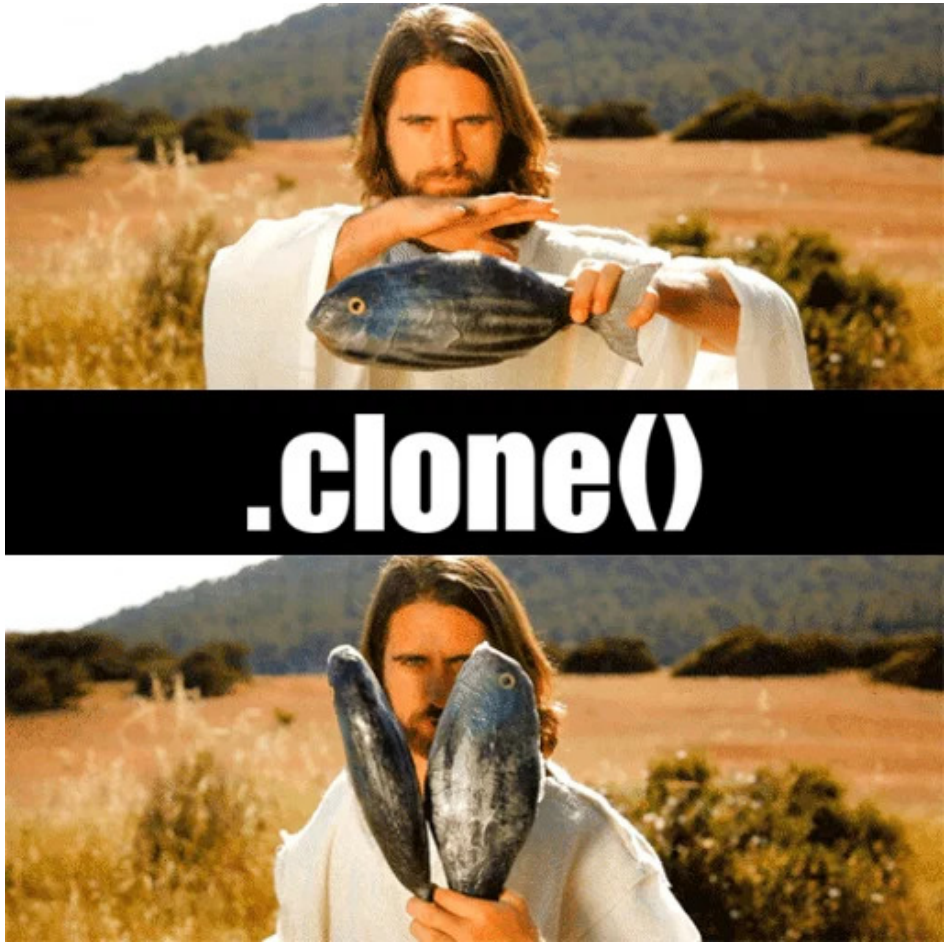
- nécessite souvent de préciser le type souhaité

Copy and Clone

- Clone

```
pub trait Clone: Sized {
 fn clone(&self) -> Self;
}
```

- Copy
  - pas de méthode additionnelle
  - permet l'utilisation de **memcopy**
  - transforme les **déplacements** en **copies**
- **Clone** et **Copy** sont dérivables



Plan

Traits

Traits classiques

Traits génériques et types associés

Traits génériques

- un trait peut être paramétré par un ou plusieurs types génériques
  - `trait Foo<A,B,C> { ... }`

Traits génériques

- un trait peut être paramétré par un ou plusieurs types génériques
  - `trait Foo<A,B,C> { ... }`
- exemple : **PartialEq**

```
pub trait PartialEq<Rhs = Self>
where Rhs: ?Sized,
{ fn eq(&self, other: &Rhs) -> bool; }

struct Foo(u32);
struct Bar { baz: Option<u32> };
impl PartialEq<Bar> for Foo {
    fn eq(&self, other: &Rhs) -> bool {
        Some(self.0) == other.baz
    }
}
```

PartialEq, Eq, PartialOrd, Ord

- Eq

```
trait Eq: PartialEq { }
```

- PartialOrd

```
pub enum Ordering {
    Less = -1,
    Equal = 0,
    Greater = 1,
}
fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>
```

- Ord : ordre total : on enlève l'option

Types associés

- type précisé lors de l'implémentation d'un trait
  - permet d'éviter de paramétrer un trait par un type
    - dans le cas où un seul type fait sens pour le trait

Types associés

- type précisé lors de l'implémentation d'un trait
  - permet d'éviter de paramétrer un trait par un type
    - dans le cas où un seul type fait sens pour le trait
- exemple : **Deref** et **DerefMut**

```
pub trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}
```

Types associés

- type précisé lors de l'implémentation d'un trait
  - permet d'éviter de paramétrer un trait par un type
    - dans le cas où un seul type fait sens pour le trait
- exemple : **Deref** et **DerefMut**

```
pub trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}
```

  - trait appliqué automatiquement par le compilateur

```
fn increment(x: &u32) -> u32 { x + 1 }
let b = Box::new(3);
assert_eq!(increment(&b), 4);
```

  - troublant pour les débutants
  - les pointeurs intelligents (**Box**, ...) l'implémentent

Add, Sub, Div, Mul, ...

- on peut mélanger types associés et traits génériques
- exemple : pour surcharger les opérateurs arithmétiques
  - +, -, *, /, ...

```
pub trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
```

- comment comprendre la définition du trait ?