1_Traits_fr

May 2, 2023

1 Traits

Un `trait` permet de spécifier au compilateur un ensemble de fonctionnalités qui est partagé par plusieurs types. Les traits permettent de partager un comportement à un niveau abstrait. Ensuite, les bornes de trait (*trait bounds*) peuvent être utiliser pour spécifier qu'un paramètre de type possède un certain comportement.

Ce comportement consiste en un ensemble de méthodes qui peuvent être appelées sur un type. Des types différents peuvent partager le même comportement si il est possible d'appeler les mêmes méthodes dessus. Les traits sont un moyens de grouper la signature de ces méthodes en ce sens.

[ ]:
```
trait Summary {
    fn summarize(&self) -> String;
}
```

Un trait peut déclarer plusieurs méthodes, les signatures sont simplements listées les unes à la suite des autres. Chaque signature se termine par un point-virgule.

1.1 Implémenter un trait pour un type

[ ]:
```
fn main() {
    trait Summary {
        fn summarize(&self) -> String;
    }

    struct NewsArticle {
        headline: String,
        location: String,
        author: String,
        content: String,
    }

    impl Summary for NewsArticle {
        fn summarize(&self) -> String {
            format!("{}, by {} ({})", self.headline, self.author, self.location)
        }
    }
```

```
    struct Tweet {
        username: String,
        content: String,
        reply: bool,
        retweet: bool,
    }

    impl Summary for Tweet {
        fn summarize(&self) -> String {
            format!("{}: {}", self.username, self.content)
        }
    }

    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know,␣
 ↪people"),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
main();
```

Une restriction à connaitre sur les traits, est qu'un n'est possible d'implémenter un trait pour type donné que si soit le trait lui-même, soit le type est initialement déclaré dans ce paquet (*crate*). Cette propriété s'appele la *orphan rule*, nommée ainsi parce que le type parent n'est pas présent. Cette règle assure ainsi que un paquet tiers ne peut pas casser le code d'un autre en implémentant un trait extérieur. Sans cette règle, deux paquets pourraient implémenter le même trait pour le même type de deux manières différentes.

1.2 Implémentations par défaut

Parfois il est utile de pouvoir définir un comportement par défaut pour quelques unes (ou toutes) les méthodes d'un trait, au lieu de laisser le champ livre à chaque implémentation pour chaque type. Il est toujours possible de remplacer l'implémentation par défaut par un autre plus spécifique lorsque le trait est implémenté pour un type précis.

[ ]:
```
fn main() {
    trait Summary {
        fn summarize_author(&self) -> String;

        fn summarize(&self) -> String {
            // implémentation par défaut, on peut appeler summarize_author ici
            // même si la définition exacte n'est pas encore connue
            format!("(Read more from {}...)", self.summarize_author())
        }
```

```
    }

    struct Tweet {
        username: String,
        content: String,
        reply: bool,
        retweet: bool,
    }

    // seule summarize_author doit être implémentée
    // summarize est l'implémentation par défaut
    impl Summary for Tweet {
        fn summarize_author(&self) -> String {
            format!("@{}", self.username)
        }
    }

    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know,␣
 ↪people"),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
main();
```

1.3 Syntaxe des bornes de trait

En Rust, la définition de types peut être paramétrée avec des paramètres de type. Nous avons déjà vu quelques exemples: `Option<T>` et `Result<T,E>`. Ces deux types en particulier n'imposent pas de contraintes sur leurs paramètres `T` et `E`, mais il est possible de le faire :

[ ]:
```
fn main() {
    trait Summary {
        fn summarize(&self) -> String;
    }

    struct Tweet {
        username: String,
        content: String,
        reply: bool,
        retweet: bool,
    }
```

```
    impl Summary for Tweet {
        fn summarize(&self) -> String {
            format!("{}: {}", self.username, self.content)
        }
    }

    struct Content<T: Summary> { // le paramètre T doit implémenter le trait␣
↪Summary
        inner: T
    }

    impl <T: Summary> Content<T> {    // Même contrainte pour cette␣
↪implémentation
        fn print_summary(&self) {
            println!("{}", self.inner.summarize());
        }
    }

    let my_content = Content {
        inner: Tweet {
            username: String::from("horse_ebooks"),
            content: String::from("of course, as you probably already know,␣
↪people"),
            reply: false,
            retweet: false,
        }
    };
    my_content.print_summary();
}
main();
```

Il possible de faire de même pour une fonction générique :

[ ]:
```
fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}

fn notify2(item: impl Summary) { // <- sucre syntaxique pour la généricité
    println!("Breaking news! {}", item.summarize());
}
```

Plusieurs contraintes peuvent être spécifiées, pour indiquer qu'un paramètre de type doit implémenter plusieurs traits :

[ ]:
```
/// T doit implémenter Summary ET Copy
fn notify<T: Summary + Copy>(item: T) {
    println!("Breaking news! {}", item.summarize());
```

```
}
```

Dans ces examples, la fonction `notify` prend possession (*ownership*) du paramètre `item`. Nous pourrions ré-écrire cet exemple avec une référence à la place :

[ ]:
```
fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

Dans un tel cas, Rust fournit un syntaxe abrégée :

[ ]:
```
/// item est une référence sur quelque chose qui implémente Summary
fn notify2(item: &dyn Summary) { // <-- mot clé dyn
    println!("Breaking news! {}", item.summarize());
}

fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know,␣
 ↪people"),
        reply: false,
        retweet: false,
    };
    notify1(&tweet);
    notify2(&tweet);
}
main();
```

La forme `impl Trait` peut aussi être utilisé pour le type de retour de la fonction pour spécifier que le type implémente un trait, mais sans spécifier exactement lequel.

[ ]:
```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from("of course, as you probably already know,␣
 ↪people"),
        reply: false,
        retweet: false,
    }
}
```

En utilisant `impl Summary` comme type de retour, la fonction `returns_summarizable` spécifie que la valeur retournée par la fonction doit nécessairement implémenter ce trait, mais sans spécifier exactement le type concret.

Cette fonctionnalité est particulièrement utile dans le cas de fonctions qui renvoie des fermetures (*closure*) ou des iterateurs, auquel cas le type concret n'est pas forcément exprimable. En effet,

les fermetures et les itérateurs produisent des types connus du compilateur seul et/ou très longs à specifier.

Une contrainte importante est que avec cette syntaxe, la fonction toujours cependant renvoyer le même type concret. Dans notre exemple, il n'est pas possible pour la fonction de renvoyer soit `NewsArticle`, soit `Tweet`. Il faut que cela soit toujours le même type. Le compilateur doit en effet pouvoir connaitre le type concret à la compilation.

Lorsque les contraintes deviennent complexes, il est possible de les spécifier dans une clause `where` à la fin de la signature de la fonction :

[ ]:
```
use std::fmt::Display;
use std::fmt::Debug;

fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,     // <-- clause where
          U: Clone + Debug
{ 0 }
```

1.4 Utiliser les bornes pour implémenter de manière conditionnelle

Comme déjà vu un peu plus haut, il est possible de spécifier des contraintes sur les paramètres de type des blocks `impl`. Ce mécanisme peut être utilisé pour implémenter de manière conditionnelle des méthodes en fonction des contraintes sur ces paramètres de type.

[ ]:
```
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {  // <-- pas de contrainte sur T
    fn new(x: T, y: T) -> Self {
        Self {
            x,
            y,
        }
    }
}

impl<T: Display + PartialOrd> Pair<T> { // <-- les méthodes ici ne sont␣
 ↪définies pour Pair<T>
                                        // <-- seulement si T implémente␣
 ↪Display et PartialOrd
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
```

```
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

1.5 Traits standards

De même, il est possible de n'implémenter un trait pour un type que si certaines contraintes sont respectées. Par exemple, dans la librairie standard, le trait `ToString` est implémenté pour tout les types `T` tel que `T` implémente déjà `Display`.

```
impl<T: Display> ToString for T {
    // --snip--
}
```

Ainsi, nous pouvons appeler la méthode `to_string` définie par le trait `ToString` sur n'importe quel type qui implémente `Display`. Par exemple il est possible de transformer en `String` n'importe quel nombre entier comme ils implémentent `Display`.

[ ]:
```
let s = 3.to_string();
```

1.6 Surcharge d'opérateurs

En Rust, beaucoup d'opérateurs peuvent être surchargées par l'implémentation de traits particuliers. Comme dans d'autres langages, ces opérateurs peuvent réaliser des opérations sur des types utilisateurs dès lors qu'ils implémentent ces traits. Ceci est possible car les opérateurs en Rust ne sont que du sucre syntaxique pour l'appel à des méthodes. Par exemple, l'opérateur `+` dans l'expression `a + b` appele en réalité la méthode `add` (`a.add(b)`). La méthode `add` est définie par le trait `Add`. Ainsi l'opérateur `+` peut être utilisé pour tout type qui implémente le trait `Add`.

[ ]:
```
use std::ops;

#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

impl ops::Add<Point> for Point {
    type Output = Point;

    fn add(self, p: Point) -> Point {
        Point{
           x: self.x + p.x,
           y: self.y + p.y,
        }
    }
```

```
}

fn main() {
    let a = Point{x: 10, y: 10};
    let b = Point{x: 20, y: 20};
    println!("Translated point = {:#?}", a + b);
}
main();
```

2 Exercises

Exercise 1: Créez deux structure *Rectangle* et *Triangle*, chacun avec une longueur et une hauteur de type `f32`. Ecrivez le code pour calculer l'aire de chacune. Décidez sur l'utilisation de trait et/ou de généricité.

[ ]:
```
fn main() {

}
main();
```

Solution 1:

[ ]:
```
struct Rectangle {
    length: f64,
    height: f64
}

struct Triangle {
    length: f64,
    height: f64
}

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Rectangle {
    fn area(&self) -> f64 { self.length * self.height }
}

impl HasArea for Triangle {
    fn area(&self) -> f64 { self.length * self.height / 2.0}
}

fn print_shape_area_static<T: HasArea>(shape: &T) {
    println!("area = {}", shape.area());
}
```

```rust
fn print_shape_area_dynamic(shape: &dyn HasArea) {
    println!("area = {}", shape.area());
}

fn main() {
    let rectangle = Rectangle { length: 5.0, height: 4.0 };
    let triangle = Triangle  { length: 3.0, height: 4.0 };
    print_shape_area_static(&rectangle);
    print_shape_area_static(&triangle);
    print_shape_area_dynamic(&rectangle);
    print_shape_area_dynamic(&triangle);
}
main();
```

Exercice 2: Simplifiez le code suivant en utilisant de la généricité

[ ]:
```rust
fn largest_i32(list: &[i32]) -> i32 {  // trouve le plus grand i32 dans la slice
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> char { // trouve le plus grand char dans la␣
 ↪slice
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);
```

```
    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
}
main();
```

Astuce: : Utilisez les traits `std::cmp::Ord` et `std::marker::Copy`.

Solution :

[ ]:
```
use std::cmp::Ord;

fn largest<T: Copy + Ord>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_idiomatic<T: Copy + Ord + Default>(list: &[T]) -> T {
    list.iter().copied().max().unwrap_or_default()
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_idiomatic(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_idiomatic(&char_list);
    println!("The largest char is {}", result);
}
main();
```