4_Closures_fr

May 2, 2023

1 Fermetures (*closures*), fonctions anonymes capturant leurs contextes

Les fermetures (*closures*) en Rust sont des fonctions anonymes qui peuvent être sauvegardées dans des variables et passées comme arguments à d'autres fonctions. Vous pouvez les créer à un endroit et les utiliser (évaluer) dans un autre contexte.

La syntaxe et les capacité des fermetures font qu'elles sont très faciles à utiliser dans beaucoup de contexte. Appeler une fermeture se fait de la même façon que l'apple à une fonction. Cependant, le typage des paramètres et du type de retour peuvent être inférés par le compilateur.

Quelques caractéristiques des fermetures :

- Les paramètres sont déclarés entre `|  |`
- Le corp est une expression (rappel: un block `{}` est une expression)
- Elles peuvent capturer les variables de leur environnement

[ ]:
```
fn main() {
    // Incrément en utiliser des fonctions et des fermetures
    fn  function (i: i32) -> i32 { i + 1 }

    // les fermetures sont anonymes
    // ici nous les assignons à des variables pour les utiliser
    let closure_annotated = |i: i32| -> i32 { i + 1 }; // avec annotations de␣
 ↪type
    let closure_inferred  = |i     |          i + 1  ; // avec inférence de type

    let i = 1;
    println!("function: {}", function(i));
    println!("closure_annotated: {}", closure_annotated(i));
    println!("closure_inferred: {}", closure_inferred(i));

}
main();
```

Les fermetures peuvent capturer leur environnement, contrairement aux fonctions.

[ ]:
```
fn main() {
    let value = 3;
```

```
    let f = |x| x + value; // <-- la fermeture fait référence à value
    println!("{}", f(2));
}
main();
```

Par défaut, les fermetures capturent leur environnement comme des références (immutables).

[ ]:
```
fn main() {
    let value = String::from("Hello !");
    let tell = || { println!("{}", value); };
    tell();
    println!("value is still here: {}", value);
}
main();
```

Les fermetures peuvent également capturer des références mutables. Dans ce cas, la variable qui contient fermeture doit être déclarée mutable.

[ ]:
```
fn main() {
    let mut value = 0;
    {
        let mut inc = || { value += 1; };
        inc();
    }
    println!("value = {}", value);
}
main();
```

Le *borrow-checker* s'assure de la sûreté des référence mutables, même pour les fermetures.

Parfois il peut être utile que la fermeture prenne la propriété (*ownership*) de leur environnement. Pour faire cela, il convient d'utiliser le mot clé `move` devant la définition de la fermeture. La propriété de toutes les variables référencées dans la fermeture sera transféré à la fermeture.

[ ]:
```
fn main() {
    let a = String::from("Hello ");
    let b = String::from("World !");
    let tell = move || { print!("{}", a); }; // here a is moved to the closure

    tell();
    println!("{}", b); // this is OK
    // println!("{}", a); // this is an error
}
main();
```

Les fermetures permettent de tirer parti de l'inférence de type et de ne pas spécifier explicitement le typage des paramètres et du type de retour. Le compilateur va bien sûr inférer un type concret.

[ ]:
```rust
fn generate_workout(intensity: u32, random_number: u32) {
    let expensive_closure = |num| {
        println!("calculating slowly...");
        // fait ici quelque chose qui prend du temps
        num
    };

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_closure(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_closure(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_closure(intensity)
            );
        }
    }
}
```

1.1 Stockage des fermetures et traits `Fn`

Remarquez que dans le code précédent, la fermeture dans `expensive_closure` va être exécutée plusieurs fois, ce qui n'est pas forcément nécessaire, d'autant plus que celle-ci est sensée prendre du temps.

Pour éviter cela, nous pouvons construire une structure qui va contenir à la fois la fermeture et le résultat de celle-ci. Cette structure va exécuter la fermeture quand le résultat est requis, et le stocker en cache pour le réutiliser. Ce pattern est souvent appelé mémoisation, ou *lazy evaluation*.

Pour définir des structures, enumérations ou fonctions qui stockent ou utilisent des fermetures, nous devons utiliser la programmation générique et les contraintes de traits.

Toutes les fermetures implémentent au moins un des trois traits : `Fn`, `FnMut` et `FnOnce`. Ceux-ci sont définis par la librairie standard.

[ ]:
```rust
// structure stockant la fermeture et son résulat
struct Cacher<T>
    where T: Fn(u32) -> u32
{
    calculation: T,
```

```
    value: Option<u32>,
}

impl<T> Cacher<T>
    where T: Fn(u32) -> u32
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            },
        }
    }
}
```

La structure `Cacher` a un membre `calculation`, dont le type est le paramètre de type `T`. La contrainte de trait indique que `T` doit implémenter le trait `Fn` (avec une certaine signature). Une fermeture que l'on voudrait stocker dans ce membre devrait avoir un paramètre de type `u32` et retourner une valeur de type `u32`.

Avant que l'on exécuter la fermeture, le membre `value` est à `None`. Lorsque du code appele `Cacher` pour demander le résultat, `Cacher` va exécuter la fermeture, et le mettre en cache dans le membre `value` avant de retourner la valeur. Si `Cacher` dispose déjà de la valeur, celle-ci est retournée sans ré-exécuter la fermeture.

> **Note:** Les fonctions peuvent aussi implémenter un des 3 traits (ou tous) `Fn`. Si ce que l'on veut faire ne demande pas de capturer l'environnement, nous pouvons utiliser une fonction à la place d'une fermeture dès que ce qui est attendu doit implémenter un trait `Fn`.

[ ]:
```
struct Cacher<T>
    where T: Fn(u32) -> u32
{
    calculation: T,
    value: Option<u32>,
}

impl<T> Cacher<T>
```

```
    where T: Fn(u32) -> u32
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            },
        }
    }
}

fn generate_workout(intensity: u32, random_number: u32) {
    // using the Cacher struct
    let mut expensive_result = Cacher::new(|num| {
        println!("calculating slowly...");
        // do something here that takes time
        num
    });

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_result.value(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_result.value(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_result.value(intensity)
            );
        }
```

```
    }
}

fn main()
{
    let mut c = Cacher::new(|a| a);

    let v1 = c.value(1);
    generate_workout(v1, 7);

}
main();
```

Au lieu de sauvegarder la fermeture dans une variable directement, nous créons une instance de `Cacher` qui va contenir cette fermeture. Puis, à tous les endroits où nous avons besoin du résultat de la fermeture, nous appelons la méthode `value` sur l'instance de `Cacher`. Cette méthode peut être appelée autant de fois que nécessaire (voire pas du tout) et le calcul coûteux représenté par la fermeture ne se fera au maximum qu'une seule fois.

Il y a une limitation à cette implémentation, elle part de l'hypothèse que c'est toujours la même valeur de paramètre `arg` qui est passé à la méthode `value`, et elle n'accepte que des fermetures qui ont exactement la signature `(u32) -> u32`

1.2 Capture de l'environnement

Les fermetures peuvent capturer des valeurs de leur environnement de trois façons, qui correspondent exactement aux trois façons dont une valeur peut être passée en paramètre : 1) en prenant la propriété, 2) en empruntant de manière mutable, et 3) en empruntant de manière non-mutable. Ces 3 façons s'encode sous la forme des 3 traits `Fn` :

- `FnOnce` consomme les variables capturées de l'environnement (*closure's environment*). Pour que ces variables capturées puissent être consommées, la fermeture doit pendre la propriété de ces variables et les déménager (*move*) dans le contexte de la fermeture. Le mot `Once` dans le nom représente le fait que cette fermeture ne peut être exécutée qu'une seule fois, comme son exécution consomme les variables capturées.
- `FnMut` peut changer son environnement en empruntant de manière mutable les variables de son contexte.
- `Fn` emprunte son environnement de manière non-mutable.