2_Smart pointers_fr

May 2, 2023

1 Smart pointers

Un pointeur est un concept général pour une variable qui contient une adresse mémoire. Cette adresse, réfère à (ou pointe vers) d'autres données. Le type de pointeur le plus courant en Rust est la *référence*.

Les pointeurs intelligents (*smart pointers*), à l'inverse, sont des structures de données qui se comportent comme des pointeurs mais portent égalements des métadonnées et/ou ont des capacités, propriétés supplémentaires. En Rust, les références sont des pointeurs qui ne font que emprunter (*borrow*) de la donnée. A l'inverse, dans bien des cas, les smart pointers possèdent (*own*) les données sur lesquelles ils pointent. Les objets `String` et `Vec` sont déjà des pointeurs intelligents dans cette définition.

Les pointeurs intelligents sont généralement implémentés sous la forme de structure. Une caractéristique qui les différencie des autres structure est leur implémentation des traits `Deref` et `Drop`.

Beaucoup de librairies définissent leurs propres types de pointeurs intelligents, et vous pouvez également écrire les votres. Les usages les plus courants sont cependant supportés directement par la librairie standard.

- `Box<T>` pour allouer sur le tas (*heap*)
- `Rc<T>` pour *reference counted* qui permet un système de propriétaires multiples.
- `Ref<T>` et `RefMut<T>`, accédés à travers `RefCell<T>`, un type qui implémente les règles d'emprunt (*borrow*), mais à l'excution

1.1 `Box<T>`

`Box` (boîte) permet de stocke le contenu sur le tas (*heap*) au lieu de sur le pile (*stack*). Ce qui reste sur la pile est le pointeur vers les données sur le tas.

Ce type est très utile dans les situations suivantes :

- Lorsqu'un type n'a pas une taille connue à la compilation mais que l'on veut utiliser une valeur de ce type dans un contexte où un taille exacte est attendue (par exemple passer une fermeture (*closure*) par valeur).
- Lorsque l'on a un grand volume de données dont on a la propriété et que l'on veut transférer la propriété (*ownership*), en s'assurant qu'il n'y aura pas d'opération de copie.
- Lorsque l'on a la propriété (*ownership*) d'une valeur, mais que l'on ne veut pas connaitre le type exacte, juste qu'il implémente une trait particulier.

[ ]:
```rust
fn main() {
    let b = Box::new(5); // stocke la valeur 5 sur le tas
    println!("b = {}", b);
} // box est désallouée lorsque l'on sort du contexte (scope)
```

A la compilation, Rust doit connaitre la taille exacte en mémoire des types. Un exemple de type dont la taille ne peut pas être connu à la compilation est un type récursif, qui peut contenir pour une part un autre instance de lui-même ainsi que d'autres données. Comme cette imbrication peut potentiellement continuer à l'infini, la taille du type ne peut pas être connue. Mais `Box` a toujours une taille connue. En l'insérant dans la définition de notre type récursif, le problème est résolu.

Le type *cons list*

Chaque élément dans une **cons list** contient deux éléments : la valeur de l'élément courant et l'élément suivant. Le dernier élément de la liste ne contient qu'une valeur spéciale (`Nil`).

```rust
// Mauvais exemple de définition de la liste, ne compile pas
// ce type est directement récursif
enum List {
    Cons(i32, List),
    Nil,
}
```

[ ]:
```rust
// Définition correct de la liste, indirectement récursive avec `Box`.
enum List {
    Cons(i32, Box<List>), // la taille d'un i32 plus l'espace pour stocker le␣
 ↪pointeur de `Box`
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1,
        Box::new(Cons(2,
            Box::new(Cons(3,
                Box::new(Nil))))));
}
```

utilisation des pointeurs intelligents comme des références avec `Deref`

[ ]:
```rust
fn main() {
    let x = 5;
    let y = &x;

    println!("x = {}", x);
    println!("y points to the value: {}", *y);
}
main();
```

[ ]:
```
fn main() {
    let x = 5;
    let y = Box::new(x); // Box<T> s'utilise comme une référence

    println!("x = {}", x);
    println!("y points to the value: {}", *y); // fonctionne car `Box` implémente `Deref`
}
main();
```

1.2 Définition de pointeurs intelligents

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

fn main() {
    // MyBox est un tuple struct qui contient un T
    struct MyBox<T>(T); // T peut être n'importe quoi

    impl<T> MyBox<T> {
        fn new(x: T) -> MyBox<T> {
            MyBox(x)
        }
    }

    // implémente Deref pour utiliser MyBox comme une référence classique
    impl<T> Deref for MyBox<T> {
        type Target = T;  // type associé à Deref

        fn deref(&self) -> &T {
            &self.0
        }
    }
    let x = 5;
    let y = MyBox::new(x);

    println!("x = {}", x);
    println!("y points to the value: {}", *y); // Rust execute ce code: *(y.deref())
}
main();
```

Sans le trait `Deref`, le compilateur ne sait déréférencer que des références classiques (`&T`). La méthode `deref` donne au compilateur la capacité à déréférencer n'importe quel pointeurs intelligent.

Coercion implicite avec `Deref`

La coercion implicite avec `Deref` est une facilité offerte par Rst sur les arguments passés aux fonctions et méthodes. La coercion implicite avec `Deref` permet la convertion implicite d'un type

qui implémente `Deref` vers le type cible de l'implémentation. Cette convertion est automatique lorsque la valeur est passée en argument d'une fonction ou d'une méthode. Une séquence d'appels à `deref` est alors générée pout convertir le type réel passé en type attendu par la fonction ou la méthode.

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

fn hello(name: &str) {
    println!("Hello, {}!", name);
}

fn main() {
    struct MyBox<T>(T);

    impl<T> MyBox<T> {
        fn new(x: T) -> MyBox<T> {
            MyBox(x)
        }
    }

    impl<T> Deref for MyBox<T> {
        type Target = T;

        fn deref(&self) -> &T {
            &self.0
        }
    }
    let m = MyBox::new(String::from("Rust"));
    hello(&m); // autrement il aurait fallu écrire: hello(&(*m)[..]);
}
main();
```

Dans cet exemple, nous appelons la fonction `hello` en passant `&m` comme argument. Le type de `m` est `MyBox<String>`. Comme le trait `Deref<Target=T>` est implémenté pour le type `MyBox<T>`, Rust peut transformer la référence `&MyBox<String>` en `&String` en appelant `deref`. De plus, la librairie standard fournit une implémentation de `Deref<Target=&str>` pour `String`, retournant ainsi une valeur `&str` (slice de chaîne de caractères). Rust peut alors appeler `deref` une nouvelle fois pour transformer `&String` en `&str`, ce qui correspond au type attendu par la fonction `hello`.

Rust effectue automatiquement une coercion avec `deref` dans les cas suivants :

- De `&T` vers `&U` quand `T: Deref<Target=U>`
- De `&mut T` vers `&mut U` quand `T: DerefMut<Target=U>`
- De `&mut T` vers `&U` quand `T: Deref<Target=U>`

Rust peut aussi automatiquement utiliser la coercion pour convertir une référence mutable en référence non-mutable. (l'inverse n'est pas possible).

[ ]:
```rust
use std::ops::Deref;
use std::fmt::Debug;

fn hello(name: &str) {
    println!("Hello, {}!", name);
}

fn main() {
    struct MyBox<T: Debug>(T);

    impl<T> MyBox<T>
    where T: Debug {
        fn new(x: T) -> MyBox<T> {
            MyBox(x)
        }
    }

    impl<T> Deref for MyBox<T>
    where T: Debug {
        type Target = T;

        fn deref(&self) -> &T {
            &self.0
        }
    }

    impl<T> Drop for MyBox<T>
    where T: Debug {
        fn drop(&mut self) {
            println!("Dropping pointer with data `{:?}`!", self.0);
        }
    }
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
main();
```

1.3 `Rc<T>`, le pointeur intelligent avec compteur de références

Dans la majorité des cas, le propriétaire d'une valeur est clair (*ownership*) : on sait exactement quelle variable est propriétaire d'une valeur. Il existe cependant des cas où une valeur peut avoir plusieurs propriétaires (*owner*). Par exemple, dans une structure de données sous forme de graph, plusieurs arrêtes peuvent pointer sur le même noeud et celui-ci appartient conceptuellement à toutes ces arrêtes en même temps. Un noeud ne devrait pas être collecté (libérer la mémoire) tant qu'au moins une arrête pointe sur lui.

Pour supporter ce cas de propriétaires multiples, Rust fournit le type `Rc<T>`, une abbréviation de *reference counting*. Le type `Rc<T>` garde la trace du nombre de références sur la valeur, ce

qui détermine si celle-ci est toujours utilisée. Lorsque ce nombre atteint zéro, la valeur peut être désallouée sans qu'aucune référence ne devienne invalide.

Nous utilisons le type `Rc<T>` lorsque nous voulons allouer sur le tas de la donnée qui sera utilisée par plusieurs partie de notre programme, mais nous ne savons pas à la compilation quelle partie finira son usage le premier.

Considérons l'exemple suivant :

[ ]:
```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    // augmente le nombre de référence de 1 à 2, a et b partagent la propriété␣
 ↪de la donnée dans Rc<List>
    let b = Cons(3, Rc::clone(&a));
    // augment le nombre de référence à 3
    let c = Cons(4, Rc::clone(&a));
}
main();
```

Lorsque nous créons `b`, au lieu de prendre la propriété (*ownership*) de `a`, nous clonons la valeur `Rc<List>` que `a` contient, et se faisant, nous augmentons le nombre de références vers la liste et laissons `a` et `b` partager la propriété des données. Nous clonons aussi à la création de `c`, augmentant encore de 1 le nombre de références. A chaque fois que nous utilisons `Rc::clone`, le compteur de références s'incrémente.

Dans l'exemple suivant, nous affichons le nombre de références à chaque changement :

[ ]:
```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("compte après la création de a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("compte après la création de b = {}", Rc::strong_count(&a));
    {   // inner scope around list c
```

```
        let c = Cons(4, Rc::clone(&a));
        println!("compte après la création de c = {}", Rc::strong_count(&a));
    }
    println!("compte après que le contexte de c se termine = {}", Rc::
 ↪strong_count(&a));
}
main();
```

1.4 `RefCell<T>` et le pattern de mutabilité interne

La mutabilité interne (*interior mutability*) est un patron de conception (*design pattern*) en Rust dans lequel nous autorisons la mutation de données alors même qu'il existe des références non-mutables sur celles-ci; une action normalement interdite par les règles d'emprunt (*borrowing*).

Avec le type `Box<T>`, les invariants des règles d'emprunt sont vérifiés à la *compilation*. Avec le type `RefCell<T>`, ces invariants sont vérifiés à l'*exécution*. Le type `RefCell<T>` est utile lorsque vous êtes sûr que le code vérifie ces invariants mais que le compilateur ne peut pas le prouver. `RefCell<T>` ne s'utilise que dans le contexte d'un seul fil d'exécution (*single thread*).

[ ]:
```
use std::cell::RefCell;
fn main() {
   let c = RefCell::new(5);
   let m = c.borrow();
   println!("{}", *c.borrow());
   // plusieurs emprunts non-mutables sont autorisés
   // un nombre de référence non-mutables est maintenu à chaque nouvel emprunt
   let p = c.borrow();
}
main();
```

[ ]:
```
use std::cell::RefCell;
fn main() {
   let c = RefCell::new(5);
   // le nombre de références doit être 0 lors d'un emprunt mutable
   let k = c.borrow_mut();
   // un seul emprunt mutable est autorisé
   // la ligne suivante produit une erreur à l'exécution
   let l = c.borrow();
}
main();
```

[ ]:
```
fn main() {
    let x = RefCell::new(vec![1,2,3,4]);

    let v = x.borrow();
    println!("{:?}", *v);
```

```
    let mut my_ref = x.borrow_mut(); // impossible d'emprunter de manière␣
 ↪mutable
    my_ref.push(1);
}
main();
```

Avec `RefCell<T>`, nous utilisons les méthodes `borrow` et `borrow_mut`, qui font partie de la partie sûre (*safe*) de l'API de `RefCell<T>`. La méthode `borrow` retourne un pointeur intelligent de type `Ref<T>`, alors que la méthode `borrow_mut` retourne un pointeur intelligent de type `RefMut<T>`. Ces deux types implémentent tous deux `Deref` et peuvent donc être traités comme des références normales.

`RefCell<T>` garde le compte du nombre de `Ref<T>` et `RefMut` actifs. A chaque appel à `borrow`, `RefCell<T>` augment le nombre d'emprunts non-mutables. De même que les règles d'emprunt vérifiées à la *compilation*, `RefCell<T>` nous laisse prendre plusieurs références non-mutables en même temps, mais une seule référence mutable à l'exclusion de toutes les autres.

Exercise 1: Quelles assertions s'appliquent à quels types (Box, Rc, RefCell) ?

- autorise plusieurs propriétaires pour la même donnée
- a un seul propriétaire
- permet un emprunt non-mutable vérifié à la *compilation*
- permet un emprunt mutable vérifié à la *compilation*
- permet un emprunt non-mutable vérifié à l'*exécution*
- permet un emprunt mutable vérifié à l'*exécution*

Solution:

Petit résumé pour bien choisir entre `Box<T>`, `Rc<T>`, et `RefCell<T>`:

- `Rc<T>` permet d'avoir plusieurs propriétaires, `Box<T>` et `RefCell<T>` n'en ont qu'un seul.
- `Box<T>` permet de vérifier à la compilation les emprunts non-mutables et mutables; `Rc<T>` permet uniquement les emprunts non-mutables vérifiés à la compilation; `RefCell<T>` permet les emprunts non-mutables et mutables vérifiés à l'exécution.
- Comme `RefCell<T>` permet les emprunts mutables vérifiés à l'exécution, il est possible de muter la valeur à l'intérieur de `RefCell<T>`, alors même que l'objet `RefCell<T>` est lui-même immutable.