2_Iterators_fr

May 2, 2023

1 Itérateurs

Les itérateurs vous permettent d'effectuer une opération sur chacun des éléments d'une séquence. L'itérateur lui-même est responsable de la logique pour itérer et de déterminer quand la séquence se termine.

En Rust, les itérateurs sont fainéants (*lazy*), ils n'ont pas d'effet à moins que vous n'appeliez une méthode qui consomme l'itérateur.

[ ]:
```
fn main() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter(); // création d'un itérateur
}
main();
```

[ ]:
```
fn main() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    for val in v1_iter { // utilisation de l'itérateur
        println!("Got: {}", val);
    }
}
main();
```

La méthode `next` retourne un élément à la fois de l'itérateur, encapsulé dans la variante `Some` de `Option`. S'il n'y a plus d'éléments, `Option::None` est retourné.

[ ]:
```
fn main() {
    let v1 = vec![1, 2, 3];

    let mut v1_iter = v1.iter();
    match v1_iter.next() {
        Some(&value) => println!("Vector element: {:?}", value),
        None => (),
    }
}
```

```
main();
```

Notez que les valeurs retournées par l'itérateur lors de l'appel à `next`, sont en fait des références non-mutables sur les valeurs du vecteur. La méthode `iter` produit un itérateur de références non-mutables.

Si nous voulons créer un itérateur qui prend la propriété (*ownership*) de `v1` et retourne les valeurs à l'intérieur du vecteur, et non plus des références à celles-ci; il nous faut utiliser `into_iter`.

De manière similaire, si nous voulons itérer sur des références mutables, il faut appeler la méthode `iter_mut` au lieu de `iter`.

Les méthodes qui appelent `next` sont appelées des adapteurs consommateurs (*consuming adaptors*) parce que les invoquer consomme l'itérateur. Un example est la méthode `sum` qui prend la propriété de l'itérateur et itère sur les éléments en appelant de manière répétée `next` afin de calculer la somme.

[ ]:
```
fn main() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum(); // l'annotation de type est ici requise

    println!("Total: {}", total);
}
main();
```

La méthode `any` est une méthode de l'itérateur qui prend une fonction en entré et retourne `true` si au moins un élément satisfait le prédicat représenté par la fonction passée.

[ ]:
```
fn main() {
    let vec1 = vec![1, 2, 3];
    let vec2 = vec![4, 5, 6];

    // `iter()` produit des `&i32`
    println!("2 in vec1: {}", vec1.iter().any(|&x| x == 2));
    // `into_iter()` produit des `i32`
    println!("2 in vec2: {}", vec2.into_iter().any(|x| x == 2));
}
main();
```

La méthode `find` des itérateurs prend également en entré une fonction et retourne le premier élément qui satisfait le prédicat représenté par la fonction, encapsulé dans une `Option`.

[ ]:
```
fn main() {
    let vec1 = vec![1, 2, 3];
    let vec2 = vec![4, 5, 6];

    // `iter()` produit des `&i32`.
```

```
    let mut iter = vec1.iter();
    // `into_iter()` produit des `i32`.
    let mut into_iter = vec2.into_iter();

    // Une référence vers le le produit de l'itérateur est `&&i32`. déstructure␣
 ↪en `i32`.
    println!("Find 2 in vec1: {:?}", iter.find(|&&x| x == 2));
    // Une référence vers le le produit de l'itérateur est `&i32`. déstructure␣
 ↪to `i32`.
    println!("Find 2 in vec2: {:?}", into_iter.find(|&x| x == 2));
}
main();
```

1.1 Méthodes qui produisent d'autres itérateurs

D'autres méthodes définies sur le trait `Iterator`, connues comme des adaptateurs (*iterator adaptors*), permettent de changer l'itérateur en un itérateur d'un autre type. Il est possible de chaîner ces méthodes pour réaliser des opérations complexes avec un code lisible. Comme tous ces itérateurs sont fainéants (*lazy*), il nous faut toujours appeler une méthode consommatrice à la fin pour récupérer le résultat.

Adaptation avec `map`

[ ]:
```
let v1: Vec<i32> = vec![1, 2, 3];

let v: Vec<_> = v1.iter().map(|x| x + 1).collect();
```

- La méthode `map` produit un itérateur dont les éléments sont ceux de l'itérateur original, transformés par la fonction passée à `map`.
- La méthode `collect` consomme l'itérateur et collecte les éléments produits dans un vecteur.

Dans cet exemple, le vecteur `v` contiendra tous les éléments du vecteur initial, incrémentés de 1.

La méthode `enumerate` produit un nouvel itérateur qui compte les éléments produits. Ce nouvel itérateur produit des tuples `(i, val)` où `i` est l'index d'itération et `val` est la valeur de l'itérateur original. https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.enumerate

[ ]:
```
fn main() {
    let a = ['a', 'b', 'c'];

    for (index, element) in a.iter().enumerate() {
        println!("L'élément à la position {} est {}", index, element);
    }
}
main();
```

2 Exercises

Exercise 1: Ecrivez une méthode qui produit un vecteur de `GeoCoordinate` à partir d'un vecteur de latitudes et un vecteur de longitudes.

[ ]:
```
#[derive(Debug)]
struct GeoCoordinate {
    /// The latitude
    latitude: f64,
    /// The longitude
    longitude: f64
}

// Ecrivez une méthode qui produit un vecteur de GeoCoordinate
// écrire une version avec enumerate, et une autre avec zip et map
// essayer d'écrire une version en style impératif et une autre en style␣
 ↪fonctionnel

fn main() {
    // TO DO (2) Testez vos implémentations

}
main();
```

Solution:

[ ]:
```
#[derive(Debug)]
struct GeoCoordinate {
    /// The latitude
    latitude: f64,
    /// The longitude
    longitude: f64
}

fn imperative(latitudes: &[f64], longitudes: &[f64]) -> Vec<GeoCoordinate> {
    let mut points: Vec<GeoCoordinate> = Vec::new();
    for (index, element) in latitudes.iter().enumerate() {
        points.push(GeoCoordinate{latitude: *element, longitude:␣
 ↪longitudes[index]});
    }
    points
}

fn functiona_bad(latitudes: &[f64], longitudes: &[f64]) -> Vec<GeoCoordinate> {
    latitudes
        .iter()
        .enumerate()
        .map(|(index, latitude)| GeoCoordinate {
```

```
            latitude: *latitude,
            longitude: longitudes[index]
        })
        .collect()
}

fn functional_idiomatic(latitudes: &[f64], longitudes: &[f64]) ->␣
 ↪Vec<GeoCoordinate> {
    latitudes
        .iter()
        .copied()
        .zip(longitudes.iter().copied())
        .map(|(latitude, longitude)| GeoCoordinate {
            latitude,
            longitude
        })
        .collect()
}

fn main() {
    let latitudes = vec![1.0, 3.0, 5.0];
    let longitudes = vec![2.0, 4.0, 6.0];
    let result = functional_idiomatic(&latitudes, &longitudes);
    println!("{:#?}", result);
}
main();
```