Системне Програмування

З використанням мови програмування **Rust.**
Fundamentals. Iterators

Ітератори

Ітератори в Rust відіграють ключову роль у багатьох програмних конструкціях, надаючи потужний і зручний інтерфейс для роботи з колекціями даних. Вони дозволяють обробляти елементи послідовно без необхідності явно вказувати індекси або управляти лічильниками циклів. Нижче розглянемо основні особливості ітераторів у Rust, їх реалізацію та використання на практиці.

Визначення

Ітератори в Rust мають змогу оперувати елементами послідовностей, таких як array, vector, option, set, map, … та ін. без використання індексів та перевірки розміру структури. Тобто унеможливлює клас помилок так чи інакше пов’язаний з неправильним (не існуючим індексом).

Ми вже використовували ітератори, але не знали що це ітератори

```
let xs : [i32; 3] = [1, 2, 3];

for x : i32 in xs {
    println!("{}", x);
}
```

насправді це ітератор

```
let xs : [i32; 3] = [1, 2, 3];
let it: Iter<i32> = xs.iter();

for x : &i32 in it {
    println!("{}", x);
}
```

Як це працює

Iterator це трейт, який має всього один метод

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

Що таке Option?

Option - це **enum** якій має два можливих значення

```
enum Option<A> {
    Some(A),
    None,
}
```

Або є значення, або нема значення.
так, в Rust нема null, undefined, etc.

```
let x = null;
```

Що таке Option?

Тобто

```
Option<Self::Item>;
```

Дає можливість повертати
Some(x) - коли є елементи
None - коли більше нема елементів,
Оскільки нам треба якимось чином
завершувати ітерацію

Як Option використовується в ітераторі?

Немає ніякої магії,
- iterator в середині себе тримає індекс
- iterator його модифікує
- iterator перевіряє на наявність наступного елементу.

За все це відповідає реалізація ітератора

Приклад

```rust
struct Months {
    months: Vec<String>,
}
```

```rust
impl Months {
    pub fn winter() -> Months {
        Months {
            months: vec![
                "December".to_string(),
                "January".to_string(),
                "February".to_string(),
            ],
        }
    }
```

Реалізація ітератора

```
impl Iterator for Months {
    type Item = String;

    fn next(&mut self) -> Option<Self::Item> {
        if ..... < self.months.len() {
            let x = Some(self.months[self.current].clone());
            .....;
            x
        } else {
            None
        }
    }
}
```

Але магії нема - нам потрібно зберігати поточний елемент (індекс)

Реалізація ітератора

```rust
struct Months {
    months: Vec<String>,
    current: usize,
}
```

```rust
impl Iterator for Months {
    type Item = String;

    fn next(&mut self) -> Option<Self::Item> {
        if self.current < self.months.len() {
            let x = Some(self.months[self.current].clone());
            self.current += 1;
            x
        } else {
            None
        }
    }
}
```

Але магії нема -
нам потрібно зберігати **індекс** поточного елемента

Створення і використання ітератора

```
let mm : Months = Months::winter();

for m : String in mm {
    println!("{}", m);
}
```

```
December
January
February
```

Ітератор “одноразовий”

```
let mm : Months = Months::winter();

for m : String in mm {
    println!("{}", m);
}

for m : String in mm {
    println!("{}", m);
}
```

Цей код не скомпілюється

Але це можна виправити

Зберігаємо індекс в іншій структурі

```rust
struct Months {
    months: Vec<String>,
}

struct MonthsIterator<'a> {
    months: &'a Vec<String>,
    current: usize,
}
```

```rust
impl Months {
    fn iter(&self) -> MonthsIterator {
        MonthsIterator {
            months: self.months.clone(),
            current: 0,
        }
    }
}
```

Реалізація ітератора

```rust
impl Iterator for MonthsIterator<'_> {
    type Item = String;

    fn next(&mut self) -> Option<Self::Item> {
        if self.current < self.months.len() {
            let x = Some(self.months[self.current].clone());
            self.current += 1;
            x
        } else {
            None
        }
    }
}
```

І тепер

```rust
let mm : Months = Months::winter();

for m : String in mm.iter() {
    println!("{}", m);
}

for m : String in mm.iter() {
    println!("{}", m);
}
```

Але магії не існує,
фактично ми створили і використали два різних ітератора

І це все?

Звісно ні.

Маючі абстракцію яка “вміє” доставати “наступний” елемент, ми можемо реалізувати велику купу методів:

- for_each
- map
- flat_map
- filter
- zip
- collect
- count,
- max,
- all,
- any
- …

Концепція

- Роботу з ітератором можна розглядати як роботу з “контейнером” в якому щось лежить, і ми можемо маніпулювати з цим.
- Всі ці методи в якості параметра приймають функцію, яка “маніпулює” з поточним елементом

Концепція

Робота з ітератором складається з трьох стадій:

1. ініціація (створення) ітератора

```
let mm : Months = Months::new();
let it : MonthsIterator = mm.iter();
```

2. робота з ітератором (необов’язкова)

```
let it : Filter<MonthsIterator, fn(...) -> ...> =
    it.filter(|x : &String|
        x.starts_with( pat: "J")
    );
```

3. термінація (завершення)

```
it.for_each(|x : String| println!("{}", x));
```

Приклади. filter

```rust
let xs : Vec<&i32> = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    .iter() : impl Iterator<Item=&i32>
    .filter(|&x : &i32 | x % 3 == 0) : impl Iterator<Item=&i32>
    .collect::<Vec<_>>();

println!("{:?}", xs);
// [3, 6, 9]
```

Приклади. filter

```rust
let xs : Vec<&i32> = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    .iter() : impl Iterator<Item=&i32>
    .filter(|&x : &i32 | *x < 5) : impl Iterator<Item=&i32>
    .collect::<Vec<_>>();

println!("{:?}", xs);
// [1, 2, 3, 4]
```

Приклади. map

```rust
let xs : Vec<i32> = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    .iter() : impl Iterator<Item=&i32>
    .map(|&x : i32 | x * 10) : impl Iterator<Item=i32>
    .collect::<Vec<_>>();

println!("{:?}", xs);
// [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Приклади. collect vs for_each

```rust
let xs : Vec<i32> = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    .iter() : impl Iterator<Item=&i32>
    .map(|&x : i32 | x * 10) : impl Iterator<Item=i32>
    .collect::<Vec<_>>();

println!("{:?}", xs);
```

```rust
let xs : () = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    .iter() : impl Iterator<Item=&i32>
    .map(|&x : i32 | x * 10) : impl Iterator<Item=i32>
    .for_each(|x : i32 | print!("{} ", x));
```

Приклади. take, skip

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    .iter() : impl Iterator<Item=&i32>
    .skip( n: 3) : impl Iterator<Item=&i32>
    .take( n: 2) : impl Iterator<Item=&i32>
    .for_each(|x : &i32 | print!("{} ", x));
// 4 5
```

Приклади. skip_while

```
[1, 1, 2, 1, 3, 4, 5, 6, 7, 8, 9, 10]
    .iter() : impl Iterator<Item=&i32>
    .skip_while(|&x : &i32 | *x < 5) : impl Iterator<Item=&i32>
    .for_each(|x : &i32 | print!("{} ", x));
// 5 6 7 8 9 10
```

Приклади. skip_while, take_while

```
[1, 1, 2, 1, 3, 4, 5, 6, 7, 8, 9, 10]
    .iter() : impl Iterator<Item=&i32>
    .skip_while(|&x : &i32 | *x < 5) : impl Iterator<Item=&i32>
    .take_while(|&x : &i32 | *x < 10) : impl Iterator<Item=&i32>
    .for_each(|x : &i32 | print!("{} ", x));
// 5 6 7 8 9
```

Приклади. Комбінація ітераторів. Zip

```
let xs : RangeFrom<char> = 'a'..;
let ys : Range<i32> = 1.. < 10;

let zs : Zip<RangeFrom<char>, Range<i32>> =
    xs.zip(ys);

zs.for_each(|t : (char, i32) |
    println!("t:{:?}", t)
);
```

```
t:('a', 1)
t:('b', 2)
t:('c', 3)
t:('d', 4)
t:('e', 5)
t:('f', 6)
t:('g', 7)
t:('h', 8)
t:('i', 9)
```

Zip автоматично “закінчується”
Коли закінчується один з ітераторів

Приклади. Видалення дублікатів за допомогою Set.

```
let xs : [i32; 6] = [1, 2, 3, 3, 2, 1];
let ys : HashSet<&i32> = xs
    .iter() : impl Iterator<Item=&i32>
    .collect::<HashSet<_>>();

println!("{:?}", ys);
// {1, 2, 3}
```

Приклади. flatten

```rust
let xs : Vec<Vec<i32>> =
    vec![
      vec![1, 2, 3],
      vec![4, 5],
      vec![7, 8, 9, 10]
    ];
let ys : Vec<&i32> = xs
    .iter() : impl Iterator<Item=&Vec<...>>
    .flatten() : impl Iterator<Item=&i32>
    .collect::<Vec<_>>();
println!("{:?}", ys);
// [1, 2, 3, 4, 5, 7, 8, 9, 10]
```

Приклади. flat_map

```rust
let xs : Vec<i32> = vec![1, 2, 3];
let ys : Vec<i32> = xs
    .iter() : impl Iterator<Item=&i32>
    .flat_map(|&x : i32| vec![-x, x]) : impl Iterator<Item=i32>
    .collect::<Vec<_>>();
println!("{:?}", ys);
// [-1, 1, -2, 2, -3, 3]
```

Приклади. count

```rust
let xs : Vec<i32> = vec![1, 2, 3, 3, 7, 3, 6];
let x : usize = xs.iter().count();
println!("{:?}", x);
// 7
```

Приклади. sum

```
let xs : Vec<i32> = vec![1, 2, 3, 3, 7, 3, 6];
let x : i32 = xs
    .iter() : impl Iterator<Item=&i32>
    .sum::<i32>();
println!("{:?}", x);
```

Приклади. all (all match)

```rust
let xs : Vec<i32> = vec![1, 2, 3, 3, 7, 3, 6];
let x : bool = xs
    .iter() : impl Iterator<Item=&i32>
    .all(|&x : i32 | x > 0);
println!("{:?}", x);
// true
let x : bool = xs
    .iter() : impl Iterator<Item=&i32>
    .all(|&x : i32 | x < 7);
println!("{:?}", x);
// false
```

Приклади. any (any match)

```rust
let xs : Vec<i32> = vec![1, 2, 3, 3, 7, 3, 6];
let x : bool = xs
    .iter() : impl Iterator<Item=&i32>
    .any(|&x : i32 | x < 2);
println!("{:?}", x);
// true
let x : bool = xs
    .iter() : impl Iterator<Item=&i32>
    .all(|&x : i32 | x < 0);
println!("{:?}", x);
// false
```

Ітератори лініві (lazy)

Це означає, що допоки ми не використали одну з функцій

- for_each
- collect
- count
- sum
- all
- any

…

Ми нічого не робимо,
ми фактично **описуємо** наші трансформації

Ітератори економно використовують пам’ять.

Це означає:
- якщо ми маємо масив з 1.000.000 елементів і ланцюжок з 10-20 map/filter/flat_map/... то ми не створюємо проміжних результатів, а беремо кожний елемент і “пропускаємо” його через весь ланцюжок трансформацій.

Ітератори звужують Scope

```rust
let xs : RangeFrom<i32> = 1..;
let ys : Vec<i32> = xs
    .filter(|x : &i32 | x % 2 == 0) : impl Iterator<Item=i32>
    .skip_while(|&x : i32 | x < 100) : impl Iterator<Item=i32>
    .take( n: 10) : impl Iterator<Item=i32>
    .flat_map(|x : i32 | vec![-x, x]) : impl Iterator<Item=i32>
    .filter(|&x : i32 | x % 10 == 0) : impl Iterator<Item=i32>
    .collect::<Vec<_>>();
println!("{:?}", ys);
// [-100, 100, -110, 110]
```

Менше шансів зробити помилку оскільки змінні доступні тільки в специфічних операціях (в середині дужок)

Висновки

І це тільки вершина айсбергу.
Ітератори це потужний інструмент для роботи с даними в **декларативному стилі**, що означає:

Я не знаю як, але я хочу зробити це.

Всі операції проходять в декларативному стилі,
Тобто ми **даємо** функцію і використовуємо **семантично-відповідну** назву методу.

Висновки

Ітератори в Rust – це потужний і зручний інструмент для роботи з послідовностями даних. Вони дозволяють спрощувати код, підвищувати його читабельність і одночасно зберігати високу продуктивність. Лінива [lazy] природа ітераторів та їх адаптери відкривають можливості для гнучкої обробки даних, при цьому зберігаючи елегантність та ефективність.

Код з лекцій,
презентації Keynote,
PDF-файли
знаходяться на GitHub:

https://github.com/djnzx/rust-course
git@github.com:djnzx/rust-course.git