클로져와 반복자

2020.3

클로져~1

- 스코프 내의 변수를 캡쳐할 수 있는 익명 함수.
- 환경을 캡쳐 한다는 점에서 함수와 다름.
- 주로 컴비네이터와 같이 사용됨.

```
let expensive_closure = |num: u32| -> u32 { // 파라메터와 리턴 값의 타입은 추론되며
    println!("calculating slowly…");                  // 이 경우와 같이 명시할 수도 있다.
    thread::sleep(Duration::from_secs(2));
    num
};
```

클로져의 사용예

```
futures::done(create_client_handshake(client_pk, client_sk, server_pk))
    .and_then(|(session, common_key, handshake)| {
        // send handshake
        Framed::new(socket, ClientHandshakeCodec)
            .send(handshake)
        .map_err(|e| {
            Error::new(
                ErrorKind::Other,
                format!("Could not send ClientHandshake {:?}", e),
            )
        })
        .map(|socket| {
            (socket.into_inner(), session, common_key)
        })
})
```

환경 캡쳐

```
fn main() {
    let x = 4;

    let equal_to_x = |z| z == x; // x를 캡쳐해 온다.

    let y = 4;

    assert!(equal_to_x(y));
}
```

```
fn main() {
    let x = 4;

    fn equal_to_x(z: i32) -> bool { z == x }. // 컴파일 되지 않는다. x가 무언지 모름.

    let y = 4;

    assert!(equal_to_x(y));
}
```

3개의 FN TRAITS

- FnOnce : 캡쳐한 변수를 소모함.
- Fn : 캡쳐한 변수를 불변으로 빌려옴
- FnMut : 캡쳐한 변수를 가변으로 빌려옴. 따라서 원본 값을 수정할 수 있다.
- 어떤 것에 속하는지는 컴파일러가 추론한다.

Move

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

    let equal_to_x = move |z| z == x;

    println!("can't use x here: {:?}", x);

    let y = vec![1, 2, 3];

    assert!(equal_to_x(y));
}
```

```
error[E0382]: use of moved value: `x`
 --> src/main.rs:6:40
  |
4 |     let equal_to_x = move |z| z == x;
  |                      -------- value moved (into closure) here
5 |
6 |     println!("can't use x here: {:?}", x);
  |                                        ^ value used here after move
  |
  = note: move occurs because `x` has type `std::vec::Vec<i32>`, which does not
  implement the `Copy` trait
```

ITER()

```
let v1 = vec![1, 2, 3];
let v1_iter = v1.iter();
for val in v1_iter {
    println!("Got: {}", val);
}
```

```
trait Iterator {
    type Item; // 연관 타입, 이 타입이 정의되어야 트레잇이 동작함, 나중에 자세히 설명할 예정임.

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

```
fn iterator_demonstration() {
    let v1 = vec![1, 2, 3];
    let mut v1_iter = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

반복자 처리

- 반복자를 소모의 예 : sum

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

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

- 새로운 반복자를 생성하는 예 : map

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

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

환경의 캡쳐

```
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_my_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter()
        .filter(|s| s.size == shoe_size)
        .collect()
}

#[test]
fn filters_by_size() {
    let shoes = vec![
        Shoe { size: 10, style: String::from("sneaker") },
        Shoe { size: 13, style: String::from("sandal") },
        Shoe { size: 10, style: String::from("boot") },
    ];

    let in_my_size = shoes_in_my_size(shoes, 10);

    assert_eq!(
        in_my_size,
        vec![
            Shoe { size: 10, style: String::from("sneaker") },
            Shoe { size: 10, style: String::from("boot") },
        ]
    );
}
```

Custom iter

```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}

impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;

        if self.count < 6 {
            Some(self.count)
        } else {
            None
        }
    }
}
```

```
fn calling_next_directly() {
    let mut counter = Counter::new();

    assert_eq!(counter.next(), Some(1));
    assert_eq!(counter.next(), Some(2));
    assert_eq!(counter.next(), Some(3));
    assert_eq!(counter.next(), Some(4));
    assert_eq!(counter.next(), Some(5));
    assert_eq!(counter.next(), None);
}
```

성능

- 벤치마크 결과는 for loop 보다 반복자가 더 빠르게 나옴.
- 제로 오버헤드 : 일반적으로, C++ 구현은 제로-오버헤드 원리를 따릅니다: 사용하지 않는 것은, 비용을 지불하지 않습니다. 그리고 더 나아가: 사용하는 것은, 더 나은 코드를 제공할 수 없습니다.

Q & A