파이썬 프로그래머를 위한 러스트 입문

윤인도

freedomzero91@gmail.com

클로저와 소유권

앞에서 클로저를 단순히 익명 함수라고만 설명하고 넘어갔습니다. 하지만 이제 스코프와 소유권을 배웠기 때문에, 클로저에 대해 좀더 자세한 얘기를 해보려고 합니다.

클로저의 가장 큰 특징은 익명 함수를 만들고 이를 변수에 저장하거나 다른 함수의 인자로 전달할 수 있다는 것입니다.

클로저의 환경 캡처

클로저는 클로저가 선언된 스코프에 있는 지역 변수를 자신의 함수 내부에서 사용할 수 있는데, 이를 환경 캡처(Environment capture)라고 부릅니다. 클로저가 변수를 자신의 스코프 내부로 가져가는 방법은 총 3가지가 존재합니다.

- 불변 소유권 대여
- 가변 소유권 대여
- 소유권 가져가기

불변 소유권 대여

클로저 `func` 는 같은 스코프에 선언된 변수 `multiplier` 를 자신의 함수 내부에서 사용할 수 있습니다. 이때 `multiplier` 의 값은 클로저에서 사용된 이후에도 스코프 내부에서 사용이 가능합니다.

```
fn main() {
    let multiplier = 5;

    let func = |x: i32| -> i32 { x * multiplier };

    for i in 1..=5 {
        println!("{}", func(i));
    }

    println!("{}", multiplier); // 👍
}
```

가변 소유권 대여

아래 예제는 `multiplier`를 가변 변수로 선언하고, 클로저 내부에서 `multiplier`의 값을 변경시키고 있습니다. 방금 살펴본 예제와 마찬가지로 클로저 호출이 끝난 다음에도 여전히 `multiplier`에 접근이 가능합니다.

```
fn main() {
    let mut multiplier = 5;

    let mut func = |x: i32| -> i32 {
        multiplier += 1;
        x * multiplier
    };

    for i in 1..=5 {
        println!("{}", func(i));
    }

    println!("{}", multiplier); // 👍
}
```

`move` 를 사용한 소유권 이동

클로저가 환경으로부터 사용하는 값의 소유권을 가져갈 수도 있습니다. 클로저가 같은 스코프에 선언된 지역 변수의 소유권을 가져가도록 하려면 클로저의 파라미터를 선언하는 코드 앞에 `move` 키워드를 사용하면 됩니다.

```
move | param, ... | body;
```

다음 예제에서는 클로저를 리턴하는 함수 `factory`를 만들었습니다.

- 여기서 리턴되는 클로저는 `factory` 함수의 파라미터인 `factor`를 캡처해 사용합니다.
- `multiplier` 변수를 모든 클로저에서 공유할 수 있게 됩니다.

```
fn factory(factor: i32) -> impl Fn(i32) -> i32 {
    |x| x * factor
}

fn main() {
    let multiplier = 5;
    let mult = factory(multiplier);
    for i in 1..=3 {
        println!("{}", mult(i));
    }
}
```

하지만 위 코드를 컴파일하면, 아래와 같은 에러가 발생합니다.

```
error[E0597]: `factor` does not live long enough
 --> src/main.rs:2:13
  |
2 |     |x| x * factor
  |     ---     ^^^^^^ borrowed value does not live long enough
  |     |
  |     value captured here
3 | }
  |  -
  |  |
  |  `factor` dropped here while still borrowed
  |  borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `notebook` due to previous error
```

문제점:

- `factor` 변수가 클로저 안에 캡처될 때, 소유권이 `factory` 로부터 클로저로 대여됩니다.
- `factory` 함수가 종료되면 `factor` 변수의 값이 삭제됩니다.

해결 방법:

`move` 는 캡처된 변수의 소유권을 클로저 안으로 이동시킵니다.

```
fn factory(factor: i32) -> impl Fn(i32) -> i32 {
    move |x| x * factor
}

fn main() {
    let multiplier = 5;
    let mult = factory(multiplier);
    for i in 1..=3 {
        println!("{}", mult(i));
    }
}
```

> 클로저에서 `move` 를 가장 많이 사용하는 경우는 멀티스레드 혹은 비동기 프로그래밍을 작성할 때입니다.

Quiz

1.

다음 페이지의 코드에서 `inc1`과 `inc2`는 같은 `count` 변수를 캡처해서 사용합니다. 아래 코드를 수정해서, `inc1`과 `inc2`가 각각 다른 `count` 변수를 캡처하도록 만들어보세요. 코드를 실행했을 때, 다음 결과가 나오면 됩니다.

```
count: 1
count: 1
```

```rust
fn main() {
    let mut count = 0;

    let mut inc1 = || {
        count += 1;
        println!("count: {}", count);
    };

    inc1();

    let mut inc2 = || {
        count += 1;
        println!("count: {}", count);
    };

    inc2();
}
```

정답

```rust
fn main() {
    let mut count = 0;

    let mut inc1 = move || {
        count += 1;
        println!("count: {}", count);
    };

    inc1();

    let mut inc2 = move || {
        count += 1;
        println!("count: {}", count);
    };

    inc2();
}
```

2.

아래 코드가 정상적으로 실행되도록 `factory` 함수를 수정하세요.

힌트: 클로저의 타입은 `impl Fn(_) -> _` 과 같이 작성하면 됩니다.

```
fn factory() -> _ {
    let num = 5;

    |x| x + num
}

fn main() {
    println!("{}", factory()(1));
}
```

정답

```
fn factory() -> impl Fn(i32) -> i32 {
    let num = 5;

    move |x| x + num
}

fn main() {
    println!("{}", factory()(1));
}
```