세상에서 제일 쉬운 러스트 프로그래밍

7장. 구조체

윤인도

freedomzero91@gmail.com

러스트는 객체지향 프로그래밍보다는 명령형 프로그래밍에 더 가깝습니다.

- 러스트 코드는 이터레이터와 클로저를 적극적으로 사용합니다.
- 클래스가 존재하지 않는 대신 구조체 `struct` 를 사용합니다.

구조체의 정의

구조체 선언

파이썬 `Person` 클래스는 객체화 시 `name`, `age` 두 변수를 파라미터로 받아 `self.name`, `self.age` 라는 인스턴스 프로퍼티에 할당

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

러스트에서 구조체를 선언하기 위해서는 `struct` 키워드 뒤에 구조체 이름을 명시하면 됩니다.

```
#[derive(Debug)] // 콘솔 출력을 위한 Debug
struct Person {
    name: String,
    age: i32,
}
```

파이썬

```python
jane = Person("jane", 30)
jane.age += 1
print(jane.name, jane.age)
print(jane.__dict__)
```

러스트

```rust
fn main() {
    let mut jane = Person { // 프로퍼티를 변경하려면 구조체를 가변으로 선언
        name: String::from("Jane"),
        age: 30
    };
    jane.age += 1;
    println!("{} {}", jane.name, jane.age);
    println!("{:?}", jane);
}
```

연관 함수(Associated function)

구조체에 포함되어 있지만 `self`를 사용하지 않는 함수

파이썬에서 `alive = True` 라는 프로퍼티를 추가

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.alive = True
```

`new`는 연관 함수로 새로운 인스턴스를 생성하는 방법을 정의하기 위해 사용

```
#[derive(Debug)]
struct Person {
    name: String,
    age: i32,
    alive: bool, // `=true` 와 같이 기본값 설정 불가
}

impl Person {
    fn new(name: &str, age: i32) -> Self {
        Person {
            name: String::from(name),
            age: age,
            alive: true, // new에서 입력받지 않은 프로퍼티를 기본값으로 설정 가능
        }
    }
}
```

메서드

메서드 `info`, `get_older` 추가

파이썬 코드 : `self` 를 받음

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.alive = True

    def info(self):
        print(self.name, self.age)

    def get_older(self, year):
        self.age += year
```

러스트 코드 : `self`가 아닌 `&self`를 받음

→ 객체의 소유권을 넘기지 않고도 객체의 프로퍼티에 접근해야 하기 때문

```
impl Person {
    ...

    fn info(&self) {
        println!("{} {}", self.name, self.age)
    }

    fn get_older(&mut self, year: i32) {
        self.age += year;
    }
}
```

⚠️ 이때 `self` 가 borrowed 되면서 mutable 인 것에 주의!

인스턴스 프로퍼티가 변경되기 때문에 `self` 가 mutable이어야 합니다.

`get_older` 메서드를 통해 age가 3 증가

```
john = Person("john", 20)
john.info()
john.get_older(3)
john.info()
```

러스트도 동일

```
fn main() {
    let mut john = Person::new("john", 20);
    john.info();
    john.get_older(3);
    john.info();
}
```

정리하면, 구조체 안에는

- `self` 파라미터를 사용하지 않는 연관 함수
- `self` 파라미터를 사용하는 메서드

모두를 정의할 수 있습니다.

튜플 구조체

- 구조체 필드가 이름 대신 튜플 순서대로 정의되는 구조체
- 필드 참조 역시 튜플의 원소를 인덱스로 참조하는 것과 동일

```
struct Color(i32, i32, i32); // 필드 타입만 정의해주면 끝
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);

    println!("{} {}", black.0, origin.0);
}
```

트레이트(trait)

파이썬은 클래스를 상속해 공통된 메서드를 사용할 수 있지만, 러스트는 구조체의 상속이 되지 않음
대신 여러 구조체에서 공통적으로 사용할 수 있는 기능 혹은 인터페이스를 "트레이트"로 구성 가능

트레이트로 메서드 공유하기

- 먼저 파이썬에서 다음과 같이 `Person` 을 상속하는 새로운 클래스 `Student` 를 선언
- 이제 `Student` 클래스는 `Person` 클래스의 모든 메서드를 사용할 수 있음

```
class Person:
    ...


class Student(Person):
    def __init__(self, name, age, major):
        super().__init__(name, age)
        self.major = major

    def say_hello(self):
        print(f"Hello, I am {self.name} and I am studying {self.major}")
```

- 러스트는 구조체를 상속할 수 없는 대신, 메서드를 공유하는 방법인 트레이트 `trait`을 사용
- `trait` 선언에서는 메서드를 "정의"하고, 구조체에서 `impl` 키워드를 사용해 "구현"

```
trait Greet {
    fn say_hello() {}
}

impl Greet for Person {}

struct Student {
    name: String,
    age: i32,
    alive: bool,
    major: String,
}
```

```rust
impl Student {
    fn new(name: &str, age: i32, major: &str) -> Student {
        Student {
            name: String::from(name),
            age: age,
            alive: true,
            major: String::from(major),
        }
    }
}

impl Greet for Student {
    fn say_hello(&self) {
        println!("Hello, I am {} and I am studying {}", self.name, self.major)
    }
}
```

```rust
fn main() {
    let mut person = Person::new("John", 20);
    person.say_hello(); // 🤫
    person.get_older(1);
    println!("{} is now {} years old", person.name, person.age);

    let student = Student::new("Jane", 20, "Computer Science");
    student.say_hello();
}
```

실행결과

```
John is now 21 years old
Hello, I am Jane and I am studying Computer Science
```

만일 아래와 같이 기본 구현체를 변경하면 코드가 컴파일되지 않습니다. 여기서 파라미터로 `&self` 를 받고 있지만, 트레이트에 정의되는 함수는 인스턴스 프로퍼티에 접근할 수 없습니다.

```
trait Greet {
    fn say_hello(&self) {
        println!("Hello, {}", self.name);
    }
}
```

이건 OK

```
 trait Greet {
     fn say_hello() {
         println!("Hello, Rustacean!");
     }
}
```

이 문제를 해결하려면 프로퍼티에 접근할 수 있는 게터(getter) 메소드를 별도로 선언

```
...

trait Greet {
    fn name(&self) -> &str;
    fn say_hello(&self) {
        println!("Hello, {}", self.name());
    }
}

impl Greet for Person {
    fn name(&self) -> &str { // 여전히 이 부분은 직접 구현해야 함
        &self.name
    }
}

fn main() {
    let p = Person {
        name: "John".to_string(),
        age: 32,
        alive: true,
    };

    p.say_hello();
}
```

파생(Derive)

- `#[derive]` 애트리뷰트를 통해 트레이트에 대한 기본 구현을 제공
- 만일 기본 구현과 다른 동작이 필요한 경우, 트레이트를 직접 구현

파생 가능한 트레이트 목록

- `Eq`, `PartialEq`, `Ord`, `PartialOrd`: 값을 비교합니다.
- `Clone`: 복사본을 통해 `&T`에서 `T`를 생성합니다.
- `Copy`: 다른 스코프로 변수를 전달할 때 값을 이동시키지 않고 복사합니다.
- `Default`: 데이터 타입의 빈 인스턴스를 생성합니다.
- `Debug`: `{:?}` 포매터를 사용할 때의 표시 형식을 지정합니다.

Copy 트레이트

`Copy`의 경우는 값을 완전히 복사할 때 사용

```
fn i32_copy(val: i32) {
    println!("i32: {}", val);
}

fn main() {
    let my_i32 = 3;

    // 함수 호출 시 my_i32의 값이
    // "복사" 되어서 소유권이 넘어가지 않음
    i32_copy(my_i32);
    println!("{}", my_i32);
}
```

- `String`과 같은 타입은 `Copy`를 구현하지 않음
- 같은 함수를 `String` 타입으로 호출하면 컴파일 에러가 발생

```
fn string_copy(val: String) {
    println!("String: {}", val);
}
fn main() {
    let my_string = String::from("Hello");
    string_copy(my_string); // 소유권 이동
    println!("{}", my_string); // 🤯
}
```

`Copy` 트레이트를 사용할 수 있는 기본 타입

- 모든 정수 타입: `u8`, `i16`, `u32`, `i64`
- 불리언 타입: `bool`
- 모든 실수형 타입: `f32`, `f64`
- 문자 타입: `char`

Clone 트레이트

`Clone`은 명시적으로 객체를 복사하고자 할 때 사용

```
class Point:
    def __init__(self, val: int):
        self.val = val
p1 = Point(5)
p2 = p1
p1.val = 3
print(f"p1.val = {p1.val}, p2.val = {p2.val}")
```

`p1`을 수정하면 `p2`도 동시에 수정됨

러스트에서는 하나의 값을 여러 객체에서 소유할 수 없으므로 다음 코드는 컴파일되지 않음

```
struct Point {
    val: i32,
}

fn main() {
    let mut p1 = Point { val: 5 };
    let p2 = &p1;
    p1.val = 3;
    println!("p1.val = {}, p2.val = {}", p1.val, p2.val);
}
```

레퍼런스가 아닌 `p1` 자체를 `p2`에 할당할 수 없음

```
struct Point {
    val: i32,
}

fn main() {
    let mut p1 = Point { val: 5 };
    let p2 = p1; // 😱
    p1.val = 3;
    println!("p1.val = {}, p2.val = {}", p1.val, p2.val);
}
```

`Clone` 트레이트의 `clone` 메서드를 사용하자

```rust
#[derive(Clone)]
struct Point {
    val: i32,
}
fn main() {
    let mut p1 = Point { val: 5 };
    let p2 = p1.clone(); // ✨
    p1.val = 3;
    println!("p1.val = {}, p2.val = {}", p1.val, p2.val);
}
```

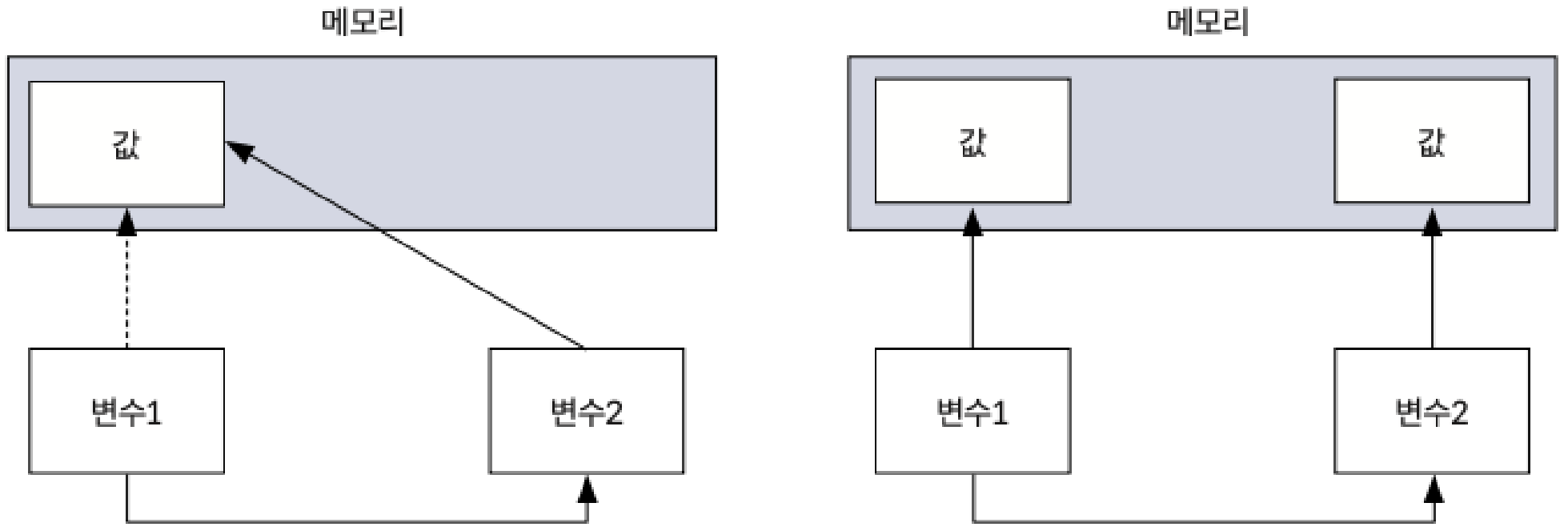

그림 7-1 Copy와 Clone 트레이트

Debug 트레이트

다음 코드는 컴파일되지 않음

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {:?}", rect1); // 🤯
}
```

에러 내용을 살펴보면 `Rectangle`을 프린트할 수 없다고 함

```
error[E0277]: `Rectangle` doesn't implement `Debug`
  --> src/main.rs:12:31
   |
12 |     println!("rect1 is {:?}", rect1); // 🤯
   |                               ^^^^^ `Rectangle` cannot be formatted using `{:?}`
   |
   = help: the trait `Debug` is not implemented for `Rectangle`
   = note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for Rectangle`
```

- 이때 컴파일러의 조언대로 `Debug` 트레이트를 파생
- `Debug`는 `{:?}` 포매터를 사용할 때의 표시 형식을 지정

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {:?}", rect1);
}
```

연습문제

1. 여기서는 계산기를 직접 구현해 볼 것입니다. 사진에서처럼 계산기는 항상 위에 현재 값을 나타내는 숫자가 있습니다.

- 계산기 위의 숫자, 즉 현재 값을 프로퍼티로 저장하고 있어야 합니다.
- 현재 값의 초기 값은 0입니다.
- 입력받은 값만큼의 더하기, 빼기를 메서드로 구현합니다.
- 더하기, 빼기는 현재 값을 직접 업데이트합니다.

샘플 코드

```
struct Calculator {
    value: i32,
}

impl Calculator {
    fn new() -> Self {
        Self { value: 0 }
    }

    fn add(&mut self, num: i32) {
        // self.value에 값을 더하세요
    }

    fn subtract(&mut self, num: i32) {
        // self.value에서 값을 빼세요
    }
}
```

정답

```
struct Calculator {
    value: i32,
}

impl Calculator {
    fn new() -> Self {
        Self { value: 0 }
    }

    fn add(&mut self, num: i32) {
        // self.value에 값을 더하세요
        self.value += num;
    }

    fn subtract(&mut self, num: i32) {
        // self.value에서 값을 빼세요
        self.value -= num;
    }
}
```