파이썬 프로그래머를 위한 러스트 입문

윤인도

freedomzero91@gmail.com

CH7. 구조체

러스트는 객체지향 프로그래밍보다는 함수형 프로그래밍에 더 가깝습니다.

- 단적인 예로 러스트 코드는 이터레이터와 클로저를 적극적으로 사용합니다.
- 이러한 이유에서 클래스가 존재하지 않습니다.
- 대신 비슷한 역할을 구조체 `struct` 를 통해서 구현할 수 있습니다.

구조체의 정의

구조체 선언

먼저 파이썬에서 클래스를 하나 정의해 보겠습니다. `Person` 클래스는 객체화 시 `name`, `age` 두 변수를 파라미터로 받고, `self.name`, `self.age` 라는 인스턴스 프로퍼티에 할당됩니다.

```
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)] // derived traits
struct Person {
    name: String,
    age: i32,
}
```

> 여기서 `#[derive(Debug)]` 는 미리 정의되어 있는 기능으로(derived trait 라고 합니다), 구조체의 내용을 보기 위해서 필요합니다.

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 {
        name: String::from("Jane"),
        age: 30
    };
    jane.age += 1;
    println!("{} {}", jane.name, jane.age);
    println!("{:?}", jane);
}
```

메소드

`alive = True` 라는 프로퍼티를 추가

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

`new`는 메소드가 아닌 연관 함수(Associated function)로 파라미터에 `self`가 들어있지 않습니다.

```
#[derive(Debug)] // derived traits
struct Person {
    name: String,
    age: i32,
    alive: bool,
}

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

인스턴스를 생성하는 메소드 말고 일반적인 메소드도 추가가 가능합니다. 먼저 파이썬에서는

```
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
```

러스트에서는 아래와 같습니다.

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

    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 주의합니다.

왜냐하면 인스턴스 프로퍼티가 변경되기 때문에 `self` 가 mutable이어야 합니다.

여기서 `info` 메소드의 `&self` 를 `self` 로 바꾸면 어떻게 될까요?

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

`get_older` 메소드를 통해 age가 3 증가합니다. 러스트에서도 동일합니다.

```
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 메소드

모두를 정의할 수 있습니다.

튜플 구조체(Tuple struct)

튜플 구조체는 구조체 필드가 이름 대신 튜플 순서대로 정의되는 구조체입니다. 필드 참조 역시 튜플의 원소를 인덱스로 참조하는 것과 동일합니다.

```
struct Color(i32, i32, i32);
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 메소드를 사용할 수 있지만, 러스트는 구조체의 상속이 되지 않습니다.

먼저 파이썬에서 다음과 같이 `Person` 을 상속하는 새로운 클래스 `Student` 를 선언합니다.

```
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}")
```

Rust는 하나의 struct를 상속하는 방법이 존재하지 않는 대신 메소드를 공유하는 방법인 `trait` 을 사용합니다.

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

```
...

impl Greet for Person {}

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

```
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();
}
```

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, Rustacean!");
    }
}
```

파생(Derive)

컴파일러는 `#[derive]` 트레이트을 통해 일부 특성에 대한 기본 구현을 제공할 수 있습니다. 보다 복잡한 동작이 필요한 경우 이러한 특성은 직접 구현할 수 있습니다.

다음은 파생 가능한 트레이트 목록입니다:

- 비교: `Eq`, `PartialEq`, `Ord`, `PartialOrd`.
- `Clone`, 복사본을 통해 `&T` 에서 `T` 를 생성합니다.
- `Copy`, '이동 시맨틱' 대신 '복사 시맨틱' 타입을 제공합니다.
- `Hash`, `&T` 에서 해시를 계산합니다.
- `Default`, 데이터 타입의 빈 인스턴스를 생성합니다.
- `{:?}` 포매터를 사용하여 값의 형식을 지정하려면 `Debug`.

다음 코드는 컴파일되지 않습니다.

```
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 합니다.

```
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 트레이트를 파생시키면 됩니다.

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

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

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