파이썬 프로그래머를 위한 러스트 입문

윤인도

freedomzero91@gmail.com

CH9. 제네릭

제네릭이란?

제네릭은 다양한 유형에 사용할 수 있는 코드를 작성할 수 있는 기능입니다. 함수, 구조체, 열거형 또는 특성을 제네릭 파라미터로 정의하면 다양한 데이터 유형에서 작동하는 재사용 가능한 코드를 만들 수 있습니다. 따라서 제네릭은 보다 유연하고 효율적인 코드를 작성하도록 도와줍니다.

타입 파라미터

제네릭이 가장 많이 사용되는 곳은 타입 파라미터입니다. 제네릭을 사용해 타입 파라미터는 `<T>` 와 같이 표시합니다.

```
fn foo<T>(arg: T) { ... }
```

다음과 같이 정수 필드를 갖는 구조체를 생각해 보겠습니다.

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
}
```

만약 `Point` 타입이 실수를 저장할 수 있도록 변경하고 싶다면 `Point` 타입을 정의하는 코드를 변경해야 합니다.

```
struct PointI32 {
    x: i32,
    y: i32,
}

struct PointF64 {
    x: f64,
    y: f64,
}

fn main() {
    let integer = PointI32 { x: 5, y: 10 };
    let float = PointF64 { x: 5.0, y: 10.0 };
}
```

불필요하게 코드가 늘어나는 것을 볼 수 있습니다. 이를 해결하기 위해 제네릭을 사용할 수 있습니다.

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 5.0, y: 10.0 };
}
```

하지만 아직 완벽하진 않습니다.

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 5.0, y: 10.0 };
    let int_float = Point { x: 5, y: 10.0 }; // 🤯
}
```

두 제네릭 타입을 받도록 고치면 됩니다.

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 5.0, y: 10.0 };
    let int_float = Point { x: 5, y: 10.0 };
}
```

> 알파벳 순서를 따라 T, U, V, X, Y, Z, ... 순으로 많이 사용하지만, 임의의 파스칼 케이스 변수명을 사용해도 상관없습니다.

메소드 정의에서도 제네릭 타입을 사용할 수 있습니다.

```
...

impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };

    let mixed = integer.mixup(float);

    println!("mixed.x = {}, mixed.y = {}", mixed.x, mixed.y);
}
```

제네릭과 트레이트

`impl Trait`

파라미터 타입

```rust
fn copy(_item: impl Copy) {
    println!("Copy");
}

fn clone(_item: impl Clone) {
    println!("Clone");
}

fn main() {
    let num = 1;
    copy(num);
    clone(num);

    let string = String::from("Hello");
    clone(string);
    // copy(string); // 🤯
}
```

트레이트 바운드

트레이트 바운드(Trait bound)란 `impl Trait` 를 사용하는 대신 좀더 간결하게 표현할 수 있는 방법입니다.

```
use std::fmt::Display;

fn some_function<T: Display>(t: &T) {
    println!("{}", t);
}

fn main() {
    let x = 5;
    some_function(&x);
}
```

이를 원래대로 `impl Trait`를 사용하면 다음과 같습니다.

```
fn some_function(t: &impl Display) {
    println!("{}", t);
}
```

트레이트 바운드를 사용하면 다음과 같이 타입을 복합적으로 표현할 수 있습니다.

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) {}
```

하지만 이러면 함수 선언을 알아보기가 어려워지기 때문에 `where` 문을 사용해 좀더 읽기 쉽게 바꿀 수 있습니다.

```
use std::fmt::{Debug, Display};

fn some_function<T, U>(t: &T, u: &U)
where
    T: Display + Clone,
    U: Clone + Debug,
{
    println!("{} {:?}", t, u);
}

fn main() {
    let x = 5;
    let y = vec![1, 2, 3];
    some_function(&x, &y);
}
```

리턴 타입

리턴 타입으로 `impl Trait` 구문을 사용하면 특정 트레이트를 구현하고 있는 타입을 리턴하도록 헐 수도 있습니다:

```
fn double(vector: Vec<i32>) -> impl Iterator<Item = i32> {
    vector.into_iter().map(|x| x * 2)
}

fn main() {
    for num in double(vec![1, 2, 3]) {
        println!("{}", num);
    }
}
```

터보피쉬

터보 피쉬 신택스는 제네릭 타입인 파라미터에 구체적인 타입을 지정하는 데 사용됩니다.

```
identifier::<type>
```

타입 어노테이션 대신에 사용되는 경우

간결성을 위해 명시적 타입 어노테이션 대신에 사용됩니다.

컴파일러가 대부분의 상황에서 타입을 추론 가능

```
use std::collections::HashMap;

fn main() {
    let mut students = HashMap::new();
    students.insert("buzzi", 100);
}
```

이런 경우는 어떤 원소를 넣는지 알 수 없기 때문에 타입을 명시적으로 알려줘야 함

```
use std::collections::HashMap;

fn main() {
    let mut students: HashMap<&str, i32> = HashMap::new();
    // students.insert("buzzi", 100);
}
```

이 경우 터보피시를 사용해서 타입 어노테이션을 대체 가능

```
use std::collections::HashMap;

fn main() {
    let mut students: HashMap = HashMap::<&str, i32>::new();
    // students.insert("buzzi", 100);
}
```

복잡한 예제

```
fn double<T>(vector: Vec<T>) -> impl Iterator<Item = T> {
    vector.into_iter().map(|x| x)
}

fn main() {
    let nums = double(vec![1, 2, 3]).collect::<Vec<i32>>();
    println!("{:?}", nums);
    let nums: Vec<String> =
        double(vec!["1".to_string(), "2".to_string(), "3".to_string()]).collect();
    println!("{:?}", nums);
}
```

명시적 타입 어노테이션이 작동하지 않을 때

```
fn main() {
    let nums: Vec<i32> = ["1", "2", "three"]
        .iter()
        .filter_map(|x| x.parse().ok())
        .collect();
}
```

`nums` 의 타입을 지정하더라도 여전히 타입 추론이 불가능합니다.

```
fn main() {
    let nums: bool = ["1", "2", "three"]
        .iter()
        .filter_map(|x| x.parse().ok())
        .collect() // 🤯
        .contains(&1);
}
```

이런 경우는 터보피쉬를 사용해야만 합니다.

```
fn main() {
    let nums: bool = ["1", "2", "three"]
        .iter()
        .filter_map(|x| x.parse().ok())
        .collect::<Vec<i32>>()
        .contains(&1);
}
```

미니프로젝트: `cat` 만들어보기

`clap`은 러스트에서 CLI 앱을 쉽게 만들 수 있도록 도와주는 크레이트입니다. 최근 릴리즈에서 `derive` 라는 기능을 사용해 앱을 더 쉽게 만드는 기능이 추가되었습니다. 이 기능을 사용하기 위해서는 설치 시 `--features derive` 옵션을 추가하면 됩니다.

```
cargo add clap --features derive
```

제일 먼저 커맨드라인 정보를 읽어올 `Args` 구조체를 선언합니다.

```
use clap::Parser;

#[derive(Parser, Debug)]
#[command(author, version, about, long_about = None)]
struct Args {
    #[arg(short, long)]
    name: String,
}
```

그 다음 파일로부터 데이터를 읽어올 함수 `cat` 을 정의합니다.

```
fn cat(filename: &str) -> io::Result<()> {
    let file = File::open(filename)?;
    let reader = BufReader::new(file);

    for line in reader.lines() {
        println!("{}", line?);
    }

    Ok(())
}
```

`cat` 함수가 제대로 작동하는지 테스트해 보겠습니다. 현재 경로에 `test.txt` 파일을 만들고 아래 내용을 입력하세요.

```
name: John
age: 32
rating: 10
```

이제 메인 함수에서 `cat`을 호출합니다.

```
use std::{
    fs::File,
    io::{self, BufRead, BufReader},
};

fn cat(filename: &str) -> io::Result<()> {
    let file = File::open(filename)?;
    let reader = BufReader::new(file);

    for line in reader.lines() {
        println!("{}", line?);
    }

    Ok(())
}

fn main() {
    cat("text.txt").unwrap()
}
```

이제 사용자로부터 정보를 입력받기 위해 처음에 만든 `Args` 구조체를 사용합니다.

```
fn main() {
    let args = Args::parse();

    cat(&args.name).unwrap()
}
```

원래는 바이너리를 사용해야 하지만, 편의를 위해 만들어진 바이너리에 옵션을 넘기는 `--` 파이프를 사용합니다.

```
cargo run -- --name my_best_friends.txt
```

실행 결과

```
name: John
age: 32
rating: 10
```

라이프타임과 스태틱

레퍼런스 그리고 소유권 대여 규칙에서 다루지 않은 한 가지가 있습니다. 바로 러스트의 모든 레퍼런스는 유효한 범위인 라이프타임이 있다는 것입니다. 대부분의 경우, 레퍼런스의 라이프타임은 변수의 타입이 추론되는 것과 마찬가지로 대부분의 상황에서 컴파일러가 추론 가능합니다.

라이프타임(lifetime)

하지만 몇몇 상황의 경우, 컴파일러에게 어떤 레퍼런스가 언제까지 유효(living)한가를 명시적으로 알려줘야 할 때가 있습니다. 예를 들어 아래와 같은 경우는 컴파일되지 않습니다.

```
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

내부 스코프에서 참조된 `x`가 스코프를 벗어나면 값이 삭제되기 때문에 `r`이 가리키고 있는 값이 없는 상태가 됩니다. 이러한 경우를 댕글링 레퍼런스(Dangling reference)라고 합니다.

아쉽게도 변수에 라이프타임을 추가하는 문법은 아직 러스트에 존재하지 않습니다. 대신 함수에서 파라미터와 리턴 값의 라이프타임을 추가하는 방법을 알아보겠습니다.

함수에서의 라이프타임

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

실행 결과

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:9:33
  |
9 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the
  signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
9 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |           ++++     ++          ++          ++
```

이 함수가 `x` 혹은 `y` 중 어떤 값을 리턴할 지 알 수 없습니다. 즉 `x`와 `y`가 언제까지 스코프에서 유효한지를 알 수 없기 때문에 리턴되는 스트링 슬라이스 역시 언제까지 유효한지를 알 수 없습니다. 따라서 리턴되는 값이 언제까지 유효한지를 알려줘야 합니다.

```
&i32        // a reference
&'a i32     // a reference with an explicit lifetime
&'a mut i32 // a mutable reference with an explicit lifetime
```

이 규칙에 따라 `longest` 에 라이프타임을 나타내면 다음과 같습니다.

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

> 라이프타임에 대해서 기억해야 할 가장 중요한 점은 "라이프타임 표기는 레퍼런스의 실제 라이프타임을 바꾸지 않는다" 라는 것입니다. 여러 레퍼런스의 라이프타임 사이의 관계를 나타냅니다.

이번에는 서로 다른 라이프타임을 갖는 `string1`과 `string2`를 사용해 보겠습니다.

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str()); // 🤯
    }
    println!("The longest string is {}", result);
}

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

`string2`의 레퍼런스가 스코프 안에서만 유효하기 때문에 이와 같은 라이프타임을 갖는 `result`는 스코프 밖에서 유효하지 않습니다.

스태틱(static) 라이프타임

한 가지 특별한 라이프타임이 있습니다. 바로 `static`으로, 해당 레퍼런스가 프로그램이 실행되는 동안 계속해서 존재할 수 있음을 나타냅니다. 모든 문자열 리터럴은 스태틱 라이프타임을 가지고 있습니다.

```
let s: &'static str = "Long live the static!";
```

이 문자열의 값은 프로그램의 바이너리에 직접 저장되어 항상 사용할 수 있습니다. 따라서 모든 문자열 리터럴의 수명은 스태틱입니다.

참고로, 문자열 관련 코드를 작성하다가 레퍼런스 관련 오류가 발생하면 오류 메시지에서 스태틱 라이프타임을 사용하라는 컴파일러의 제안을 볼 수 있습니다. 하지만 라이프타임은 문자열의 존재 기간을 명확하게 명시하는 용도이기 때문에 바로 스태틱 라이프타임을 사용하지 말고, 이 문자열의 정확한 라이프타임을 먼저 적용하는 것이 중요합니다.