세상에서 제일 쉬운 러스트 프로그래밍

9장. 제네릭과 라이프타임

윤인도

freedomzero91@gmail.com

제네릭(Generic)

- 함수, 구조체, 트레이트 등을 만들 때 특정 타입에 대해서만 구현하는 것이 아닌, 타입 파라미터로 나타내는 여러 타입에 동시에 구현할 수 있도록 하는 러스트의 강력한 기능
- 여러 기능을 제네릭 파라미터로 정의하면 하나의 코드로 여러 타입에서 작동하는 **재사용 가능한** 코드를 만들 수 있음
- 제네릭은 보다 유연하고 효율적인 코드를 작성 가능하게 함

간단한 파이썬 코드

```
def bigger(num1, num2):
    return num1 if num1 > num2 else num2
print(bigger(2, 7))
print(bigger(2.0, 7.0))
print(bigger(2.0, 7))
```

타입 힌트를 추가하면 🤯

```
def bigger(num1: int | float, num2: int | float) -> int | float:
    return num1 if num1 > num2 else num2
```

러스트에서는 다른 타입의 파라미터를 받을 수 없음!

```
fn bigger(a: i32 | f64, b: i32 | f64) -> i32 | f64{
    if a > b {
        a
    } else {
        b
    }
}
fn main() {
    println!("{}", bigger(2, 7));
    println!("{}", bigger(2.0, 7.0));
    println!("{}", bigger(2.0, 7));
}
```

결국 다음과 같이 함수 3개를 만들어야 합니다.

```
fn bigger_i32(a: i32, b: i32) -> i32 {
    ...
}

fn bigger_f64_i32(a: f64, b: i32) -> f64 {
    ...
}

fn bigger_f64(a: f64, b: f64) -> f64 {
    ...
}
```

이러한 문제를 해결할 수 있는 것이 바로 제네릭! (뒤에서 설명합니다)

```rust
fn bigger<T: PartialOrd, U: PartialOrd + Into<T>>(a: T, b: U) -> T {
    let b = b.into();
    if a > b {
        a
    } else {
        b
    }
}
fn main() {
    println!("{}", bigger(2, 7));
    println!("{}", bigger(2.0, 7.0));
    println!("{}", bigger(2.0, 7));
}
```

타입 파라미터

- 제네릭을 사용해 타입 파라미터는 `<T>`와 같이 표시
- `T`라는 것은 `i32`와 같이 정해진 타입이 아닌 임의의 타입을 의미

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

정수 필드를 갖는 구조체

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

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

만약 `Point` 타입이 실수를 저장할 수 있도록 변경하고 싶다면?

```
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 해결하기 위해 제네릭을 사용

```
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.

메서드 정의에서도 제네릭 타입을 사용 가능

```
...

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);
}
```

실행 결과

```
mixed.x = 5, mixed.y = 4
```

제네릭과 트레이트

아쉽게도 제네릭은 만능이 아닙니다.

```rust
fn add<T, U>(x: T, y: U) -> T {
    x + y
}
fn main() {
    let x = add(1, 2);
    let y = add(1.0, 2.0);
    let z = add(1, 2.0);
    println!("x = {}, y = {}, z = {}", x, y, z);
}
```

```
error[E0369]: cannot add `U` to `T`
 --> src/main.rs:2:7
  |
2 |     x + y
  |     - ^ - U
  |     |
  |     T
  |
help: consider restricting type parameter `T`
  |
1 | fn add<T: std::ops::Add<U, Output = T>, U>(x: T, y: U) -> T {
  |         ++++++++++++++++++++++++++++++
```

제네릭 타입에 특성이라는 제한 조건을 추가하기 위해서 트레이트를 사용

파라미터 타입

`impl Trait`을 함수의 파라미터에 추가해서 특정 트레이트를 구현하는 타입으로 제한

```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); // 🤯
}
```

실행 결과

```
 error[E0277]: the trait bound `String: Copy` is not satisfied
   --> src/main.rs:16:10
    |
 16 |     copy(string); //
| ---- ^^^^^^ the trait `Copy` is not implemented for `String` ||
| required by a bound introduced by this call
|
 note: required by a bound in `copy`
   --> src/main.rs:1:21
    |
 1  | fn copy(_item: impl Copy) {
    |                     ^^^^ required by this bound in `copy`
```

트레이트 바운드(Trait bound)

- `impl Trait` 를 사용하는 대신 좀더 간결하게 표현할 수 있는 방법
- 타입 `T` 는 반드시 `Display` 트레이트를 구현한 타입이어야 함

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

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

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

`impl Trait`를 사용한 코드와 비교

```
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) {}
```

`where` 문을 사용해 좀더 읽기 쉽게 바꿀 수 있다!

```
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);
}
```

터보피시

터보피시는 제네릭 타입인 파라미터에 구체적인 타입을 지정하는 데 사용

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

타입 애너테이션 대신 사용되는 경우

- 간결성을 위해 명시적 타입 애너테이션 대신에 사용

컴파일러는 다음과 같이 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();
}
```

이 경우 터보피시를 사용해서 타입 애너테이션을 대체 가능

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

fn main() {
    let mut students: HashMap = HashMap::<&str, i32>::new();
}
```

복잡한 예제: 터보피시 또는 타입 애너테이션!

```
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 애너테이션이 작동하지 않을 경우

타입 애너테이션 덕분에 컴파일되는 코드

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

- 하지만 `nums`의 타입을 지정하더라도 여전히 타입 추론이 불가능
- `collect`의 결과를 추론할 수 없기 때문

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

중간 변수를 지정해도 되지만...

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

```rust
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 크레이트
- 최근 릴리즈에서 `derive` 라는 기능을 사용해 앱을 더 쉽게 만드는 기능이 추가
- 이 기능을 사용하기 위해서는 설치 시 `--features derive` 옵션을 추가

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

제일 먼저 커맨드라인 정보를 읽어올 `Args` 구조체를 선언

```rust
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 정의

```
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` 함수를 테스트하기 위해 현재 경로에 `test.txt` 파일을 만들고 아래 내용을 입력

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

이제 메인 함수에서 `cat`을 호출

```rust
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("test.txt").unwrap()
}
```

이제 사용자로부터 정보를 입력받기 위해 처음에 만든 `Args` 구조체를 사용

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

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

만들어진 바이너리에 옵션을 넘기는 `--` 파이프를 사용해 값을 전달

```
cargo run -- --name test.txt
```

실행 결과

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

라이프타임(lifetime)과 스태틱

- 러스트의 모든 레퍼런스는 유효한 범위인 라이프타임이 존재
- 라이프타임은 타입이 추론되는 것과 마찬가지로 대부분의 상황에서 컴파일러가 추론 가능

라이프타임

특정 상황에서는 컴파일러에게 어떤 레퍼런스가 언제까지 유효(living)한가를 명시해야 함

예를 들어 아래와 같은 경우는 컴파일되지 않음

```
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
        // 내부 스코프에서 참조된 `x`가 스코프를 벗어나면 값이 삭제
    }

    // `r`이 가리키고 있는 값이 없으므로
    // 댕글링 레퍼런스(Dangling reference)가 됨
    println!("r: {}", r);
}
```

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

함수에서의 라이프타임

두 개의 문자열 슬라이스 `x`와 `y`를 입력받아서 둘 중 길이가 긴 값을 리턴하는 함수

```
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 {
  |           ++++     ++          ++           ++
```

⚠️ `x`와 `y`가 유효한 스코프를 알 수 없음 → 리턴되는 스트링 슬라이스의 스코프 역시 알 수 없음

라이프타임 표기 방법

```
&i32 // 레퍼런스
&'a i32 // 명시적인 라이프타임이 추가된 레퍼런스
&'a mut i32 // 명시적인 라이프타임이 추가된 가변 레퍼런스
```

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
    }
}
```

실행 결과

```
The longest string is abcd
```

> ‘라이프타임 표기는 레퍼런스의 실제 라이프타임을 바꾸지 않는다’는 점은 반드시 기억!
> 단지 여러 레퍼런스가 갖는 각자의 라이프타임들 사이의 관계를 표시하는 것 뿐입니다.

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 않음

서로 다른 라이프타임을 명시하고 가장 오래 살아남는 `x`만 리턴하면 코드를 동작하게 할 수 있음

```
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, 'b>(x: &'a str, y: &'b str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        "y is no use here 🥲"
    }
}
```

실행결과

```
The longest string is long string is long
```

스태틱(static) 라이프타임

스태틱 라이프타임은 레퍼런스가 프로그램이 시작하는 순간부터 끝날 때까지 계속 존재함을 의미

→ 전역 변수!

로그 레벨을 나타내는 `LOG_LEVEL` 이라는 변수를 `my_module.rs` 에 선언하고, 메인 모듈에서 참조

```
pub let LOG_LEVEL: i32 = 0;
```

오류를 무시하고 `main.rs` 에 다음과 같이 작성한 후 컴파일

```
mod my_module;
use my_module::LOG_LEVEL;

fn main() {
    println!("LOG_LEVEL is {}", LOG_LEVEL);
}
```

실행 결과

```
 ...
error: expected item, found keyword `let`
 --> src/my_module.rs:2:5
  |
2 | pub let LOG_LEVEL: i32 = 0;
  |     ^^^ consider using `const` or `static` instead of `let` for global variables
...
```

가장 중요한 에러는 일반 변수를 전역 변수로 선언할 수 없다는 것

컴파일러의 조언에 따라서 `my_module.rs`를 다음과 같이 수정하고 컴파일

```
pub static LOG_LEVEL: i32 = 0;
```

실행결과

```
LOG_LEVEL is 0
```

참고1:

- 모든 문자열 리터럴은 스태틱 라이프타임을 가지고 있음
- 평상시에는 `static` 키워드를 생략할 수 있어서, 다음 두 줄은 사실상 같은 코드

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

참고2:

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

```rust
fn dummy() -> &str {
    "Long live the static!"
}
fn main() {
    println!("{}", dummy());
}
```

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:15
  |
1 | fn dummy() -> &str {
  |               ^ expected named lifetime parameter
  |
  = help: this function's return type contains a
  borrowed value, but there is no value
  for it to be borrowed from
help: consider using the `'static` lifetime, but this
is uncommon unless you're returning a borrowed value
from a `const` or a `static`
  |
1 | fn dummy() -> &'static str {
  |                +++++++
help: instead, you are more likely to want to
return an owned value
  |
1 | fn dummy() -> String {
  |               ~~~~~~
```

정말 프로그램 전체에서 사용할 값이 아니라면 바로 스태틱 라이프타임을 사용하지 말고,

- 이 문자열의 정확한 라이프타임을 먼저 적용하거나,
- `String` 타입을 대신 사용

```
fn dummy() -> String {
    "Long live the static!".to_string()
}
fn main() {
    println!("{}", dummy());
}
```

lazy_static

- 스태틱 라이프타임은 변수의 값이 컴파일 타임에 결정돼야 함
- 따라서 값이 런타임에 결정될 때는 컴파일되지 않음

```
static STATIC: i32 = define_static();

fn define_static() -> i32 {
    3
}

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

이런 경우에는 `lazy_static` 크레이트를 사용하면 편리!

```
cargo add lazy_static
```

```rust
#[macro_use]
extern crate lazy_static;

lazy_static! { // ✨
    static ref STATIC: i32 = define_static();
}

fn define_static() -> i32 {
    3
}

fn main() {
    println!("{}", *STATIC);
}
```

static vs const

컴파일이 되지 않았던 `LOG_LEVEL` 예제의 실행 결과로 돌아가서...

```
error: expected item, found keyword `let`
--> src/my_module.rs:2:5
  |
2 | pub let LOG_LEVEL: i32 = 0;
  |     ^^^ consider using `const` or `static` instead of `let` for global variables
```

`static`을 사용했지만, 컴파일러는 `const` 혹은 `static`을 사용해 전역 변수를 선언하라고 조언

같은 예제를 `const`를 사용하도록 `my_module.rs`만 수정

```
pub const LOG_LEVEL: i32 = 0;
```

실행결과

```
LOG_LEVEL is 0
```

`static`을 사용한 것과 같은 결과

그럼 두 키워드는 정확히 어떤 차이가 있는 걸까요?

1. 변수의 저장 위치
    - `static`은 변수가 스택 영역에 선언
    - `const`는 바이너리에 값이 포함돼 컴파일

    따라서 `static` 변수보다 `const` 변수에 접근하는 것이 더 빠름
2. 값에 접근하는 방법
    - `static`은 스택 영역의 메모리 공간에 단 하나의 값이 저장
    - `const`는 컴파일 타임에 `const`가 사용되는 코드의 모든 위치가 해당 값으로 변경

예를 들어, 이 코드를 컴파일하면

```
const CONST: i32 = 3;
fn main() {
    let num = 1 + CONST;
    println!("{}", CONST);
}
```

그러면 다음 코드와 같아짐!

```
fn main() {
    let num = 4; // 1 + 3
    println!("{}", 3);
}
```

결론

- 전역 변수의 고정된 메모리 주소가 필요한 경우가 아니라면, 대부분 `const`를 사용
- 다만 `const`가 너무 많이 참조된다면 바이너리 크기가 커질 수 있음