세상에서 제일 쉬운 러스트 프로그래밍

8장. 모듈과 크레이트

윤인도

freedomzero91@gmail.com

러스트의 모듈 시스템

크레이트

러스트 코드를 묶을 수 있는 가장 작은 단위

바이너리 크레이트

컴파일되어 바이너리 파일을 생성하는 크레이트

- 생성 : `cargo new <프로젝트명>`
- 엔트리포인트 : `main.rs`

```
.
├── Cargo.toml
└── src
    └── main.rs
```

라이브러리 크레이트

- 다른 크레이트나 패키지에서 코드를 참조할 수 있도록 제공
- 컴파일되지 않기 때문에 바이너리를 생성하지 않음
- 생성 : `cargo new --lib`
- 엔트리포인트 : `lib.rs`

```
.
├── Cargo.toml
└── src
    └── lib.rs
```

크레이트 루트

"컴파일 엔트리포인트"

- 바이너리 크레이트는 `src/main.rs` 파일
- 라이브러리 크레이트는 `src/lib.rs` 파일

모듈

- 파이썬에서 모듈은 파일 단위로 구분 → `*.py` 파일은 모두 모듈
- 러스트는 파일 단위로, 또는 파일 하나에서도 여러 개의 모듈을 정의할 수 있음

러스트에서 모듈 만들어보기 - 2개의 파일을 생성

```rust
// main.rs
fn main() {}
```

```rust
// my_module.rs
mod dummy1 {}
mod dummy2 {}
```

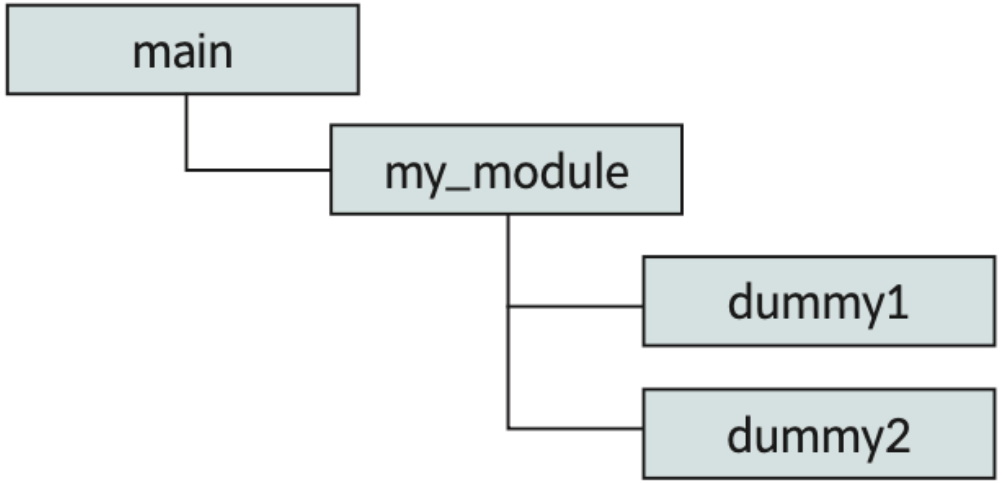

그림 8-1 크레이트 내부 모듈 구조

공개 vs 비공개

러스트의 모든 모듈과 객체는 기본적으로 비공개

외부에서 모듈에 접근하거나 모듈 내부의 객체에 접근을 허용하려면 `pub` 키워드를 사용

```
pub mod {
    // 모듈
}

pub fn {
    // 함수
}

pub struct {
    // 구조체
}

impl Struct {
    pub fn public_method(&self) {} // 공개 메서드
    fn private_method(&self) {} // 비공개 메서드
}
```

모듈 사용하기

- `use` : "특정 경로"를 "현재 스코프"로 가져오는 역할

  > ⚠️ 주의: 경로는 항상 크레이트 루트로부터 시작!

- `mod` 키워드는 해당 모듈을 사용하겠다고 선언하는 역할

`mod new_module`에 대해 컴파일러는 다음 순서대로 해당 모듈을 탐색

1. `mod new_module` 다음에 해당 모듈의 정의가 나와야 합니다.

```
mod new_module {
  fn new_func() {
    ...
  }
  ...
}
```

2. `src/new_module.rs` 파일을 찾아봅니다.
3. `src/new_module` 폴더에서 `mod.rs` 파일을 찾아봅니다.

```
pub mod new_module;
```

특정 모듈에 대한 접근은 크레이트 루트를 기준으로 절대 경로를 사용

`src/new_module.rs` 밑의 `MyType`을 찾는 방법

```
use crate::new_module::MyType
```

`self`는 현재 모듈이며, 이를 기준으로 상대 경로를 정의 가능

```
mod mod2 {
    fn func() { println!("mod2::func()"); }
    mod mod1 {
        pub fn func() { println!("mod2::mod1::func()"); }
    }
    pub fn dummy() {
        func();
        self::func();
        mod1::func();
        self::mod1::func();
    }
}
fn main() {
    mod2::dummy();
}
```

실행 결과

```
mod2::func()
mod2::func()
mod2::mod1::func()
mod2::mod1::func()
```

`super`는 현재 모듈의 상위 모듈을 의미하고 이를 기준으로 상대 경로를 정의 가능

```
mod mod1 {
    pub fn dummy() {
        println!("Hello, world!");
    }
}
mod mod2 {
    use super::mod1; // use crate::mod1 와 동일
    pub fn dummy() {
        mod1::dummy();
    }
}
fn main() {
    mod2::dummy();
}
```

실행 결과

```
Hello, world!
```

패키지

- 여러 크레이트를 모아 놓은 것
- `Cargo.toml` 파일로 해당 패키지를 빌드하는 방법을 정의
- 패키지 = 1 _ `라이브러리 크레이트` + n _ `바이너리 크레이트`

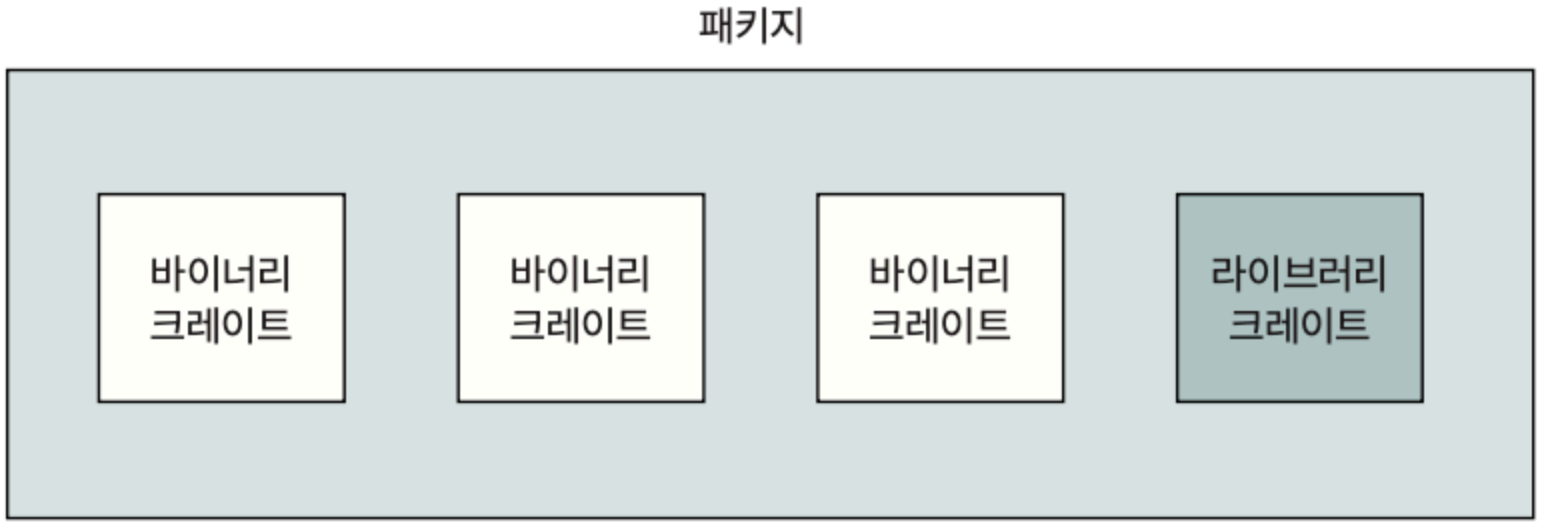

그림 8-2 패키지와 크레이트 구조

- 크레이트 루트에 `main.rs`가 있으면 바이너리, `lib.rs`가 있으면 라이브러리로 인식
- `Cargo.toml`의 `[[bin]]` 섹션에 경로를 지정해 여러 바이너리 크레이트를 추가

```
[package]
name = "my_package"
version = "0.1.0"
edition = "2018"
[lib]
name = "my_library"
path = "src/lib.rs"
[[bin]]
name = "my_binary1"
path = "src/bin/binary1.rs"
[[bin]]
name = "my_binary2"
path = "src/bin/binary2.rs"
```

이때의 폴더 구조

```
my_package
├── Cargo.toml
└── src
    ├── bin
    │   ├── binary1.rs
    │   └── binary2.rs
    └── lib.rs
```

패키지	여러 크레이트의 집합체
크레이트	라이브러리 또는 바이너리를 생성하는 모듈 집합체
모듈	구조체, 함수 등의 집합체

모듈과 크레이트 사용해보기

파이썬 폴더에 `my_modle.py` 를 생성

```
def greet():
    print(f"Hi! I am hello_bot")


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

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

이제 이 함수와 클래스를 `main.py` 에서 참조

```
from my_module import greet, Person

if __name__ == '__main__':
    greet()

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

`bots` 폴더를 만들고 `hello_bot.py` 파일을 추가

```
.
├── bots
│   └── hello_bot.py
├── main.py
└── my_module.py
```

`hello_bot.py` 는 다음과 같습니다.

```
BOT_NAME = "hello_bot"


def hello():
    print("Hello, humans!")
```

`my_module.py`에서 `greet` 함수가 `BOT_NAME` 을 이용하도록 수정

```
from bots.hello_bot import BOT_NAME


def greet():
    print(f"Hi! I am {BOT_NAME}")
```

그 다음 `main.py` 에서 `bots` 모듈을 사용

```
from bots.hello_bot import hello
from my_module import greet, Person

if __name__ == '__main__':
    hello()

    greet()

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

동일한 구조를 러스트에서 구현

`src` 폴더에 `my_module.rs` 를 생성

```
src
├── main.rs
└── my_module.rs
```

`pub` 키워드를 사용해 함수 및 구조체를 공개로 설정

```rust
pub fn greet() {
    println!("Hi! I am hello_bot");
}

pub struct Person {
    pub name: String,
    age: i32,
}

impl Person {
    pub fn new(name: &str, age: i32) -> Self {
        Person {
            name: String::from(name),
            age: age,
        }
    }
    pub fn get_older(&mut self, year: i32) {
        self.age += year;
    }
}
```

`main.rs` 에서 `my_module` 모듈을 참조

```rust
mod my_module; // src/my_module.rs에서 모듈 찾기

// 함수와 구조체 불러오기
use my_module::{greet, Person};

fn main() {
    greet();

    let mut john = Person::new("john", 20);
    john.get_older(3);
    println!("{}", john.name);
    // println!("{}", john.age); // 😱 비공개 프로퍼티
}
```

다음으로는 하위 폴더 `bots` 와 `hello_bot.rs` 와 `mod.rs` 두 파일을 생성

```
src
├── bots
│   ├── hello_bot.rs
│   └── mod.rs
├── main.rs
└── my_module.rs
```

⚠️ 중요!

하위 폴더를 모듈로 만드는 경우에는 `mod.rs` 가 있어야 해당 모듈의 정보를 읽을 수 있음
따라서 `mod.rs` 에는 `hello_bot` 모듈의 정보가 있어야 함

```
pub mod hello_bot; // hello_bot.rs를 찾아봄
```

이제 `hello_bot.rs` 파일을 작성

```
pub static BOT_NAME: &str = "hello_bot";

pub fn hello() {
    println!("Hello, humans!");
}
```

> `static` 키워드는 9장에서 설명

`BOT_NAME`을 `src/my_module.rs`에서 참조

```rust
use crate::bots::hello_bot::BOT_NAME;

pub fn greet() {
    println!("Hi! I am {}", BOT_NAME);
}
```

`main.rs` 에서 `bots` 모듈을 사용

`main.rs` 는 크레이트 루트이므로 `use bots::hello_bot::hello;`

```rust
mod my_module; // src/my_module.rs을 찾아봄
mod bots; // src/hello/mod.rs을 찾아봄

use my_module::{greet, Person};
use bots::hello_bot::hello;

fn main() {
    hello();
    greet();

    let mut john = Person::new("john", 20);
    john.get_older(3);
    println!("{}", john.name);
}
```