세상에서 제일 쉬운 러스트 프로그래밍

5장. 소유권

윤인도

freedomzero91@gmail.com

메모리 관리

- 프로그램의 데이터는 메모리에 저장됩니다.
- 컴퓨터의 메모리는 제한되어 있습니다.
- 파이썬이나 고와 같은 언어는 가비지 콜렉터를 이용해 언어 차원에서 자동으로 메모리를 관리하고, C/C++같은 언어들은 개발자가 직접 메모리를 관리합니다.

파이썬은 모든 객체의 데이터를 힙 영역에 저장하고, 가비지 콜렉션을 이용해 관리합니다.

- 사용되지 않는 객체를 주기적으로 삭제하거나, 메모리 사용량이 너무 높은 경우 객체를 삭제합니다.
- 파이썬의 코드 실행 속도가 느려지는 원인이 됩니다.
- 어떤 객체가 언제 메모리에서 할당 해제되는지를 명시적으로 알 수 있는 방법이 없습니다.

러스트는 소유권(Ownership)이라는 개념을 통해 메모리를 관리합니다.

소유권 모델 덕분에 러스트 프로그램은 메모리 안전성과 스레드 안전성이 보장됩니다.

- 메모리 안전성
  - 하나의 값에 대해서 단 하나의 코드만 접근하기 때문에 예상치 못하게 값이 변경되지 않음
  - C/C++같은 언어에서는 잘못된 포인터 사용이나 잘못된 메모리 접근과 같은 이유로 버그가 발생하거나 메모리 누수가 발생
- 스레드 안전성 : 여러 개의 스레드에서 하나의 값에 접근하고자 할 때 발생할 수 있는 경합 조건(Race condition)이나 데드락(Deadlock)이 발생하지 않음

스택과 힙

스택

- 스택 영역은 함수가 실행될 때 사용하는 메모리 공간으로, 함수에서 사용하는 지역 변수가 스택에 저장됩니다.
- 스택에 저장된 변수의 주소는 프로그램이 실행될 때 정해지기 때문에 빠르게 값에 접근할 수 있습니다.
- 함수 실행이 종료되면 스택 영역에서 사용된 모든 지역 변수는 메모리에서 삭제됩니다.

힙

- 힙 영역은 동적으로 할당되는 메모리를 위해 존재하는 공간
- 해당 메모리 공간에 할당한 변수가 더 이상 필요하지 않으면 할당 해제해주어야 합니다.

파이썬은 스택을 사용하지 않고 모든 객체를 힙 영역에 저장합니다. 이렇게 저장된 객체들은 파이썬에서 가비지 콜렉션을 통해 메모리를 관리하기 때문에 파이썬을 사용할 때는 메모리 관리에 신경쓰지 않아도 됩니다.

참고: 파이썬의 가비지 콜렉션에 대해서

러스트는 스택 영역과 힙 영역 모두를 사용합니다.

러스트는 기본적으로 아래와 같이 함수에서 사용하는 모든 값을 제한된 크기의 스택 영역에 저장합니다. 따라서 함수 호출이 종료되면 지역 변수 `foo` 와 `var` 는 모두 삭제됩니다.

```
fn foo() {
    let foo = "foo";
    let var = 5;
}
```

힙 영역은 함수에서 명시적으로 선언하는 경우에만 사용되는데, 힙 영역에 저장하는 값은 전역적으로 (globally) 접근이 가능합니다. 나중에 배울 `Box` 타입을 사용해 선언하면 됩니다.

```
fn main() {
    let num = Box::new(1);
}
```

- 함수에서 사용하는 지역 변수의 값들은 모두 스택 영역에 저장

- 전역적으로 사용되는 값들은 힙 영역에 저장

소유권 규칙 자세히 알아보기

- 모든 "값"들은 해당 값을 "소유"하고 있는 소유자(Owner)가 존재합니다.
- 한 번에 하나의 소유자만 존재할 수 있습니다. 하나의 값에 두 개의 소유자가 동시에 존재할 수 없습니다.
- 소유자가 현재 코드의 스코프에서 벗어나면, 값은 메모리에서 할당 해제됩니다.

값에 대한 소유권

러스트에서는 어떤 값이 더이상 사용되지 않는지를 소유권을 사용해 판단합니다. 모든 값에 소유자를 지정하고, 이 값을 소유하고 있는 소유자가 없게 되면 즉시 값이 메모리에서 할당 해제되는 원리입니다.

```
fn main() {
    let x = 1;
    // x가 삭제됨
}
```

같은 함수 내에서라도 스코프를 벗어나면 즉시 값은 사라집니다.

```
fn main() {
    let x = 1;
    {
        let y = x;
        println!("{} {}", x, y);
        // y가 삭제됨
    }
    println!("{} {}", x, y); // 이 라인은 컴파일되지 않음
    // x가 삭제됨
}
```

함수의 인자로 값을 전달하면, 해당 값은 함수 내부로 전달되고, 함수 내부에서는 해당 값을 소유하고 있는 소유자가 존재하게 됩니다.

```
fn dummy(x: String) {
    println!("{}", x);
    // x가 삭제됨
}

fn main() {
    let x = String::from("Hello");
    dummy(x);
    println!("{}", x);  // 이 라인은 컴파일되지 않음
}
```

그러면 모든 값은 다른 함수에 전달하면 영원히 사용하지 못하는 걸까요?

소유권 돌려주기

`"Hello"`라는 값을 소유하고 있는 변수만 `x` → `y` → `z` 순서로 바뀌고, 값은 그대로 있게 됩니다.

```
fn dummy(y: String) -> String {
    println!("{}", y);
    y
}

fn main() {
    let x = String::from("Hello");
    let z = dummy(x);
    println!("{}", z);
}
```

하지만 이 방법은 값을 전달하는 과정에서 복사가 계속 일어난다는 단점이 있습니다.

레퍼런스와 소유권 빌리기

소유권을 잠시 빌려줄 수 있는 개념인 대여(borrow)가 있습니다. 변수 앞에 `&` 키워드를 사용하면 되는데, 해당 변수의 레퍼런스(reference)를 선언한다는 의미입니다.

> 레퍼런스란 소유권을 가져가지 않고 해당 값을 참조할 수 있는 방법입니다.

```
fn main() {
    let x = String::from("Hello");
    let y = &x;

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

`y`의 빌려온 소유권은 함수가 끝나면 다시 `x`에게로 반환됩니다.

```
fn dummy(y: &String) {
    println!("{}", y);
    // 소유권이 x로 되돌아감
}

fn main() {
    let x = String::from("Hello");
    dummy(&x);
    println!("{}", x);
}
```

가변 레퍼런스

전달받은 문자열에 새로운 문자열을 추가하고 싶다면 어떻게 해야 할까요?

```
fn dummy(y: &String) {
    y.push_str(" world!"); // 😨
    println!("{}", y);
}

fn main() {
    let x = String::from("Hello");
    dummy(&x);
    println!("{}", x);
}
```

어떤 변수의 레퍼런스를 만들 때, 원래 변수가 불변이라면 레퍼런스를 사용해 원래 변수의 값을 바꿀 수 없습니다.

```
   Compiling rust_part v0.1.0 (/Users/code/temp/rust_part)
error[E0596]: cannot borrow `*y` as mutable, as it is behind a `&` reference
 --> src/main.rs:2:5
  |
1 | fn dummy(y: &String) {
  |             ------- help: consider changing this to be a mutable reference: `&mut String`
2 |     y.push_str(" world!");
  |     ^^^^^^^^^^^^^^^^^^^^^ `y` is a `&` reference, so the data it refers to
  |
  cannot be borrowed as mutable
```

컴파일러의 조언에 따라서 `y`를 가변 레퍼런스로 수정해 보겠습니다.

1. `dummy` 함수의 파라미터 `y` 의 타입을 `&mut String` 으로 변경
2. 변수 `x`를 가변 변수로 선언
3. `dummy` 함수에 `x`를 전달할 때 가변 레퍼런스 `&mut x`로 전달

```rust
fn dummy(y: &mut String) {
    y.push_str(" world!");
    println!("{}", y);
}

fn main() {
    let mut x = String::from("Hello");
    dummy(&mut x);
    println!("{}", x);
}
```

가변 레퍼런스를 사용할 때 주의해야 하는 점은 소유권 규칙의 두 번째 규칙인 "한 번에 하나의 소유자만 존재할 수 있다" 입니다.

```
fn main() {
    let mut x = String::from("Hello");
    let y = &mut x;
    let z = &mut x;

    println!("{} {}", y, z);
}
```

러스트에서는 하나의 값에 대한 여러 개의 가변 레퍼런스를 허용하지 않습니다.

```
   Compiling rust_part v0.1.0 (/Users/code/temp/rust_part)
error[E0499]: cannot borrow `x` as mutable more than once at a time
 --> src/main.rs:4:13
  |
3 |     let y = &mut x;
  |             ------ first mutable borrow occurs here
4 |     let z = &mut x;
  |             ^^^^^^ second mutable borrow occurs here
5 |
6 |     println!("{} {}", y, z);
  |                        - first borrow later used here
```

하지만 일반 레퍼런스를 여러 개 만드는 것은 문제가 없습니다.

```
fn main() {
    let x = String::from("Hello");
    let y = &x;
    let z = &x;

    println!("{} {}", y, z);
}
```

클로저와 소유권

클로저의 환경 캡처

클로저가 선언된 스코프에 존재하는 지역 변수를 자신의 내부에서 사용하는 것

- 불변 소유권 대여
- 가변 소유권 대여
- 소유권 가져가기

```rust
fn main() {
    let multiplier = 5;
    let func = |x: i32| -> i32 { x * multiplier };
    for i in 1..=5 {
        println!("{}", func(i));
    }
    println!("{}", multiplier); // 👍
}
```

실행 결과

```
5
10
15
20
25
5
```

가변 레퍼런스도 가능

```
fn main() {
    let mut multiplier = 5;
    let mut func = |x: i32| -> i32 {
        multiplier += 1;
        x * multiplier
    };
    for i in 1..=5 {
        println!("{}", func(i));
    }
    println!("{}", multiplier); // 👍
}
```

move를 사용한 소유권 이동

`factory`를 `main` 함수에서 사용해 만든 클로저를 호출하면 `multiplier` 변수를 모든 클로저에서 공유 가능

`move` 키워드를 빼고 컴파일하면 에러 발생

```
fn factory(factor: i32) -> impl Fn(i32) -> i32 {
    move |x| x * factor // move 추가됨
}

fn main() {
    let multiplier = 5;
    let mult = factory(multiplier);
    for i in 1..=3 {
        println!("{}", mult(i));
    }
}
```

연습문제

1. [소유권 돌려주기] `take_ownership` 함수를 수정해 코드가 동작하도록 만들어 보세요.

```
fn take_ownership(s: String) {
    println!("{}", s);
}

fn main() {
    let s1 = String::from("hello, world");
    let s2 = take_ownership(s1);

    println!("{}", s2);
}
```

정답

```
fn take_ownership(s: String) -> String {
    println!("{}", s);
    s
}

fn main() {
    let s1 = String::from("hello, world");
    let s2 = take_ownership(s1);

    println!("{}", s2);
}
```

2. [레퍼런스] 다음 코드가 동작하도록 고쳐보세요.

```
fn main() {
    let x = String::from("hello, world");
    let y = x;
    println!("{},{}",x,y);
}
```

정답

```
fn main() {
    let x = String::from("hello, world");
    let y = &x;
    println!("{},{}",x,y);
}
```

3. [가변 레퍼런스] 다음 코드가 동작하도록 고쳐보세요.

```
fn main() {
    // Fix error by modifying this line
    let  s = String::from("hello, ");

    borrow_object(&mut s);

    println!("Success!");
}

fn borrow_object(s: &mut String) {
    println!("{}", s);
}
```

정답

```
fn main() {
    // Fix error by modifying this line
    let mut s = String::from("hello, ");

    borrow_object(&mut s);

    println!("Success!");
}

fn borrow_object(s: &mut String) {
    println!("{}", s);
}
```

4. [환경 캡처] 다음 페이지의 코드에서 오른쪽 코드가 컴파일되지 않는 이유를 설명해 보세요.

```rust
fn main() {
    let mut name = "👴".to_string();

    let mut inc1 = || {
        name.push_str("❤️");
        println!("{}", name);
    };

    inc1();

    let mut inc2 = || {
        name.push_str("🦀");
        println!("{}", name);
    };

    inc2();
}
```

```rust
fn main() {
    let mut name = "👴".to_string();

    let mut inc1 = move || {
        name.push_str("❤️");
        println!("{}", name);
    };

    inc1();

    let mut inc2 = move || {
        name.push_str("🦀");
        println!("{}", name);
    };

    inc2();
}
```

정답

`move` 키워드로 인해 환경 캡처된 변수의 소유권이 클로저 안으로 이동되었기 때문에 이후에 해당 값을 재사용할 수 없습니다.

이런 경우 `clone` 을 사용해 값을 복제해서 해결할 수 있습니다.

```
fn main() {
    let mut name = "👴".to_string();
    let new_name = name.clone();

    let mut inc1 = move || {
        name.push_str("❤️");
        println!("{}", name);
    };

    inc1();

    let mut name = new_name;
    let mut inc2 = move || {
        name.push_str("🦀");
        println!("{}", name);
    };

    inc2();
}
```

5. [`move`] 아래 코드가 정상적으로 실행되도록 `factory` 함수를 수정하세요.

힌트: 클로저의 타입은 `impl Fn(_) -> _` 과 같이 작성하면 됩니다.

```
fn factory() -> _ {
    let num = 5;

    |x| x + num
}

fn main() {
    println!("{}", factory()(1));
}
```

정답

```
fn factory() -> impl Fn(i32) -> i32 {
    let num = 5;

    move |x| x + num
}

fn main() {
    println!("{}", factory()(1));
}
```