파이썬 프로그래머를 위한 러스트 입문

윤인도

freedomzero91@gmail.com

CH11. 스마트 포인터

포인터란?

포인터는 메모리 주소를 포함하는 변수입니다. 러스트에서 가장 일반적인 종류의 포인터는 레퍼런스입니다.

스마트 포인터란?

스마트 포인터는 포인터처럼 작동하지만, 레퍼런스의 개수 등을 포함하고 있는 자료구조입니다.

Box 타입

만일 어떤 타입의 크기를 컴파일 타임에 미리 알 수 없다면?

자기 자신을 필드값의 타입으로 갖는 재귀 형태의 구조체(Recursive type)를 정의해 보겠습니다.

```
struct Node {
    value: i32,
    next: Option<Node>,
}

fn main() {
    let mut head = Node {
        value: 1,
        next: None,
    };
    head.next = Some(Node {
        value: 2,
        next: None,
    });
    println!("{}", head.value);
}
```

`Box` 를 사용하라고 함

```
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}

fn main() {
    let mut head = Node {
        value: 1,
        next: None,
    };
    head.next = Some(Box::new(Node {
        value: 2,
        next: None,
    }));
    println!("{}", head.value);
}
```

Box<T>

`Box`가 대체 무엇일까?

`Box`를 사용하면 스택이 아닌 힙에 데이터를 저장할 수 있습니다. 스택에 남는 것은 힙 데이터에 대한 포인터입니다.

Box<T> 사용하기

아래 예제는 `Box` 를 사용하여 `i32` 값을 힙에 저장하는 방법을 보여줍니다.

```
fn main() {
    let my_box = Box::new(5);
    println!("my_box = {}", my_box);
}
```

스코프 마지막에 `my_box` 역시 삭제됨

`Box` 는 주로 다음과 같은 상황에 사용됩니다.

- 컴파일 시 크기를 알 수 없는 타입 내부의 값에 접근해야 하는 경우
- 크기가 큰 값의 소유권을 이전하고 싶지만, 메모리 효율성을 위해 전체 값이 복사되지 않도록 해야 하는 경우
- 특정 타입이 아닌, 특정 트레이트를 구현하는 타입의 변수의 소유권을 가져오고 싶은 경우

첫 번째 상황은 위에서 이미 살펴본 `Node` 의 경우입니다. 이제 나머지 각각의 경우를 자세히 살펴보겠습니다.

소유권을 효율적으로 전달하기

레퍼런스 대신

```
fn transfer_box(_data: Box<Vec<i32>>) {}

fn transfer_vec(_data: Vec<i32>) {}

fn main() {
    let data = vec![0; 10_000_000];

    transfer_vec(data.clone());

    let boxed = Box::new(data);
    transfer_box(boxed);
}
```

`dyn` 과 `Box` 로 트레이트 타입 표현하기

```
cargo add rand
```

```rust
struct Dog {}
struct Cat {}

trait Animal {
    fn noise(&self) -> &'static str;
}

impl Animal for Dog {
    fn noise(&self) -> &'static str {
        "🐶멍멍!"
    }
}

impl Animal for Cat {
    fn noise(&self) -> &'static str {
        "🐱야옹!"
    }
}
```

```
fn random_animal() -> impl Animal {
    if rand::random::<f64>() < 0.5 {
        Dog {}
    } else {
        Cat {}
    }
}

fn main() {
    for _ in 0..10 {
        println!("{}", random_animal().noise());
    }
}
```

```
error[E0308]: `if` and `else` have incompatible types
  --> src/main.rs:24:9
   |
21 | /     if rand::random::<f64>() < 0.5 {
22 | |         Dog {}
   | |         ------ expected because of this
23 | |     } else {
24 | |         Cat {}
   | |         ^^^^^^ expected struct `Dog`, found struct `Cat`
25 | |     }
   | |_____- `if` and `else` have incompatible types
   |
help: you could change the return type to be a boxed trait object
   |
20 | fn random_animal() -> Box<dyn Animal> {
   |                       ~~~~~~~      +
help: if you change the return type to expect trait objects,
box the returned expressions
   |
22 ~         Box::new(Dog {})
23 |     } else {
24 ~         Box::new(Cat {})
   |
```

```
fn random_animal() -> Box<dyn Animal> {
    if rand::random::<f64>() < 0.5 {
        Box::new(Dog {})
    } else {
        Box::new(Cat {})
    }
}
```

실행 결과

```
🐱야옹!
🐶멍멍!
🐶멍멍!
🐶멍멍!
🐱야옹!
🐱야옹!
🐶멍멍!
🐶멍멍!
🐶멍멍!
🐶멍멍!
```

`Rc<T>`

만일 하나의 값에 여러 개의 소유자를 정말로 가지고 싶다면 어떻게 할까요? 이럴 때 사용할 수 있는 자료형이 바로 `Rc<T>` 입니다. 레퍼런스 카운팅(Reference counting)의 앞 자를 따서 만든 이름으로, `Rc<T>` 역시 스마트 포인터입니다.

일반적인 레퍼런스처럼 디레퍼런스해서 사용 가능

```
use std::rc::Rc;

fn main() {
    let origin = Rc::new(1);

    assert_eq!(1, *origin);
}
```

마지막 순간까지

프로그램의 여러 부분이 읽을 수 있도록 힙에 일부 데이터를 할당하고 컴파일 시점에 어느 부분이 데이터를 마지막으로 사용할지 결정할 수 없을 때 `Rc<T>` 타입을 사용합니다.

```rust
fn main() {
    let cloned;
    {
        let origin = "Rust".to_string();
        cloned = &origin; // 🤯
    }
    println!("{}", cloned);
}
```

```rust
use std::rc::Rc;

fn main() {
    let cloned;
    {
        let origin = Rc::new(1);
        cloned = origin.clone();
    }
    println!("{}", cloned);
}
```

여기서 `clone`을 사용하면, 실제로 값이 복사되는 것이 아니라 `Rc`의 레퍼런스 카운트가 1 증가합니다.

레퍼런스 카운팅

`Rc<T>` 는 값의 소유권을 가지고 있는 변수가 몇 개인지를 계속 확인하고 있다가, 값을 소유하고 있는 변수가 전부 사라지면 값을 메모리에서 삭제합니다.

```rust
use std::rc::Rc;

fn main() {
    let origin = Rc::new(0);
    println!("Reference count: {}", Rc::strong_count(&origin));
    {
        let _dup1 = Rc::clone(&origin);
        println!("Reference count: {}", Rc::strong_count(&origin));
        {
            let _dup2 = &origin.clone();
            println!("Reference count: {}", Rc::strong_count(&origin));
        }
        println!("Reference count: {}", Rc::strong_count(&origin));
    }
    println!("Reference count: {}", Rc::strong_count(&origin));
    // origin drops here
}
```

실행 결과

```
Reference count: 1
Reference count: 2
Reference count: 3
Reference count: 2
Reference count: 1
```

> `Rc<T>`는 멀티스레드 환경에서 동작하지 않습니다. 멀티스레드 환경에서는 `Arc<T>`를 사용해야 하며, 자세한 내용은 나중에 다루겠습니다.

Quiz

```
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}

fn main() {
    let mut head1 = Node {
        value: 1,
        next: None,
    };
    let node1 = Node {
        value: 2,
        next: None,
    };
    head1.next = Some(Box::new(node1));

    let mut head2 = Node {
        value: 3,
        next: None,
    };
    head2.next = Some(Box::new(node1)); // 🤯

    println!("{} {}", head1.value, head1.next.unwrap().value);
    println!("{} {}", head2.value, head2.next.unwrap().value);
}
```

정답

```rust
use std::rc::Rc;

struct Node {
    value: i32,
    next: Option<Rc<Node>>,
}

fn main() {
    let mut head1 = Node {
        value: 1,
        next: None,
    };
    let node1 = Rc::new(Node {
        value: 2,
        next: None,
    });
    head1.next = Some(Rc::clone(&node1));

    let mut head2 = Node {
        value: 3,
        next: None,
    };
    head2.next = Some(Rc::clone(&node1));

    println!("{} {}", head1.value, head1.next.unwrap().value);
    println!("{} {}", head2.value, head2.next.unwrap().value);
}
```

`RefCell<T>`

`Rc<T>`의 한계

Rc<T>를 사용하면 프로그램의 여러 부분에서 읽기 전용으로 데이터를 공유할 수 있습니다. 하지만 Rc<T>가 불변 레퍼런스를 통해 값을 공유하기 때문에, 공유받은 값을 변경하는 것은 불가능합니다. 아래 예시를 살펴봅시다.

```rust
use std::rc::Rc;

struct Owner {
    name: String,
    tools: Rc<Vec<Rc<Tool>>>,
}

struct Tool {
    owner: Rc<Owner>,
}

pub fn main() {
    let indo = Rc::new(Owner {
        name: "indo".to_string(),
        tools: Rc::new(vec![]),
    });
    let pliers = Rc::new(Tool {
        owner: Rc::clone(&indo),
    });
    let wrench = Rc::new(Tool {
        owner: indo.clone(),
    });

    indo.tools.push(Rc::clone(&pliers)); // 🤯
    indo.tools.push(Rc::clone(&wrench));

    println!("Pliers owner: {}", pliers.owner.name);

    for tool in indo.tools.iter() {
        println!("Tool's owner: {:?}", tool.owner.name);
    }
}
```

실행 결과

```
error[E0596]: cannot borrow data in an `Rc` as mutable
  --> src/main.rs:24:5
   |
24 |     brad.tools.push(Rc::clone(&pliers)); // 🤯
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ cannot borrow as mutable
   |

...
```

```rust
use std::{cell::RefCell, rc::Rc};

struct Owner {
    name: String,
    tools: RefCell<Vec<Rc<Tool>>>,
}

struct Tool {
    owner: Rc<Owner>,
}

pub fn main() {
    let indo = Rc::from(Owner {
        name: "indo".to_string(),
        tools: RefCell::new(vec![]),
    });
    let pliers = Rc::from(Tool {
        owner: Rc::clone(&indo),
    });
    let wrench = Rc::from(Tool {
        owner: indo.clone(),
    });

    indo.tools.borrow_mut().push(Rc::clone(&pliers));
    indo.tools.borrow_mut().push(Rc::clone(&wrench));

    println!("Pliers owner: {}", pliers.owner.name);

    for tool in indo.tools.borrow().iter() {
        println!("Tool's owner: {:?}", tool.owner.name);
    }
}
```

내부 가변성(Interiror mutability)

`RefCell<T>`가 불변이어도 내부의 값은 가변으로 사용 가능

```
indo.tools.borrow_mut().push(Rc::clone(&pliers));
```

불변 소유권 대여도 가능

```
indo.tools.borrow().iter()
```

소유권 규칙

- 여러 번 빌려도 괜찮습니다
- 한 번 빌리는 것도 괜찮습니다
- 하지만 가변과 불변이 대여는 불가능합니다

런타임 시간에 소유권이 확인되기 때문에 컴파일이 되지만 런타임 에러 발생

```
use std::{cell::RefCell, rc::Rc};

struct Owner {
    name: String,
    tools: RefCell<Vec<Rc<Tool>>>,
}

struct Tool {
    owner: Rc<Owner>,
}
```

```
pub fn main() {
    let indo = Rc::from(Owner {
        name: "indo".to_string(),
        tools: RefCell::new(vec![]),
    });
    let pliers = Rc::from(Tool {
        owner: Rc::clone(&indo),
    });
    let wrench = Rc::from(Tool {
        owner: indo.clone(),
    });

    let mut borrow_mut_tools1 = indo.tools.borrow_mut();
    let mut borrow_mut_tools2 = indo.tools.borrow_mut(); // 🤯
    borrow_mut_tools1.push(Rc::clone(&pliers));
    borrow_mut_tools2.push(Rc::clone(&wrench));

    println!("Pliers owner: {}", pliers.owner.name);

    for tool in indo.tools.borrow().iter() {
        println!("Tool's owner: {:?}", tool.owner.name);
    }
}
```

실행 결과

```
thread 'main' panicked at 'already borrowed: BorrowMutError', src/main.rs:25:44
```

`Rc<RefCell<T>>`

`RefCell<T>`를 사용하는 일반적인 방법은 `Rc<T>`와 함께 사용하는 것입니다. `Rc<T>`를 사용하면 일부 데이터의 소유자를 여러 명 가질 수 있지만, 해당 데이터에 대한 불변 액세스 권한만 부여한다는 점을 기억하세요. `RefCell<T>`를 보유한 `Rc<T>`가 있다면, 여러 소유자를 가질 수 있고 변경할 수 있는 값을 얻을 수 있습니다!

언제 무엇을

	`Box<T>`	`Rc<T>`	`RefCell<T>`
소유권	한 개	한 개를 공유	한 개
소유권 확인 시점	불변/가변 소유권을 컴파일 타임에 확인	불변 소유권을 컴파일 타임에 확인	불변/가변 소유권을 런타임에 확인
특징	스코프를 벗어나면 레퍼런스도 모두 삭제	레퍼런스가 존재한다면 스코프를 벗어나도 값이 유지됨	`RefCell<T>`가 불변이어도 내부의 값은 가변으로 사용 가능

> RefCell<T>는 멀티스레드 코드에서는 작동하지 않는다는 점에 유의하세요! Mutex<T>는 스레드에 안전한 RefCell<T의 버전이며, Mutex<T에 대해서는 나중에 설명하겠습니다.

Quiz

```
use std::fmt::Display;
use std::vec::Vec;

#[derive(Debug)]
struct Node<T> {
    data: T,
    children: Vec<Node<T>>,
}

impl<T: Display> Node<T> {
    fn new(data: T) -> Node<T> {
        Node {
            data,
            children: Vec::new(),
        }
    }

    fn depth_first(&self) {
        println!("{}", self.data);
        for child in self.children.iter() {
            child.depth_first();
        }
    }
}

fn main() {
    let mut a = Node::new('A');
    let mut b = Node::new('B');
    let c = Node::new('C');
    let d = Node::new('D');

    b.children.push(d);
    a.children.push(b);
    a.children.push(c);

    a.depth_first();
}
```

아래 메소드를 추가하고 싶습니다.

```
    fn add_child(&mut self, child: Wrapper<Node<T>>) {
        self.children.push(child);
    }
```

```
fn main() {
    let a = wrap(Node::new('A'));
    let b = wrap(Node::new('B'));
    let c = wrap(Node::new('C'));
    let d = wrap(Node::new('D'));

    a.borrow_mut().add_child(Rc::clone(&b));
    a.borrow_mut().add_child(Rc::clone(&c));
    b.borrow_mut().add_child(Rc::clone(&d));
    a.borrow_mut().depth_first();
}
```

정답

```rust
use std::cell::RefCell;
use std::fmt::Display;
use std::rc::Rc;
use std::vec::Vec;

type Wrapper<T> = Rc<RefCell<T>>;

fn wrap<T>(data: T) -> Wrapper<T> {
    Rc::new(RefCell::new(data))
}

#[derive(Debug)]
struct Node<T> {
    data: T,
    children: Vec<Wrapper<Node<T>>>,
}

impl<T: Display> Node<T> {
    fn add_child(&mut self, child: Wrapper<Node<T>>) {
        self.children.push(child);
    }

    fn new(data: T) -> Node<T> {
        Node {
            data,
            children: Vec::new(),
        }
    }

    fn depth_first(&self) {
        println!("node {}", self.data);
        for child in self.children.iter() {
            child.borrow().depth_first();
        }
    }
}

fn main() {
    let a = wrap(Node::new('A'));
    let b = wrap(Node::new('B'));
    let c = wrap(Node::new('C'));
    let d = wrap(Node::new('D'));

    a.borrow_mut().add_child(Rc::clone(&b));
    a.borrow_mut().add_child(Rc::clone(&c));
    b.borrow_mut().add_child(Rc::clone(&d));
    a.borrow_mut().depth_first();
}
```