스마트 포인터

2020.3

포인터, 스마트 포인터

- 포인터
- 스마트 포인터
  - 추가적인 메타 데이터
  - 추가적인 기능
  - 데이터를 직접 소유하는 경우가 많음

String, Vector

- 참조자는 단순히 빌리는것, 스마트 포인터는 소유하면서 추가적 기능 제공.
- String, Vector는 스마트 포인터이다.
- 용량 등의 메타 데이터를 추가적으로 갖고,
- String의 경우 유효한 UTF8임을 보장하는 기능 보유.

일반적 구조체와 다른점

- Deref와 Drop trait를 (특별하게) 구현한다는 점.
- 예를 들어 Rc는 일반적 구조체가 스코프를 벗어나면 Drop되는 것에 반해서,
- 참조 카운터가 0이 되는 순간 drop되도록 (특별히) Drop을 구현함.
- Deref는 스마트 포인터 임을 신경쓰지 않고 사용할 수 있게 해준다.

다룰 내용

- Box<T>
- Rc<T>
- RefCell<T>, Ref<T>, RefMut<T>
- 내부 가변성(interior mutability)
- 참조 순환(reference cycles)

Box<T>

- 데이터를 힙에 저장하고, 그 포인터를 가짐.
- 포인터의 크기는 usize이므로 컴파일 타임에 결정 가능함.
- 데이터가 큰경우 스택보다 힙에 담아야 함.
- 나중에 trait object를 다룰 때 Box<T>를 더 논의할 예정.

재귀적 타입

- 재귀적 타입은 컴파일 타임에 크기를 알 수 없으므로 컴파일 할 수 없는데
- 포인터는 크기가 고정이므로 재귀적 타입을 사용할 수 있다.

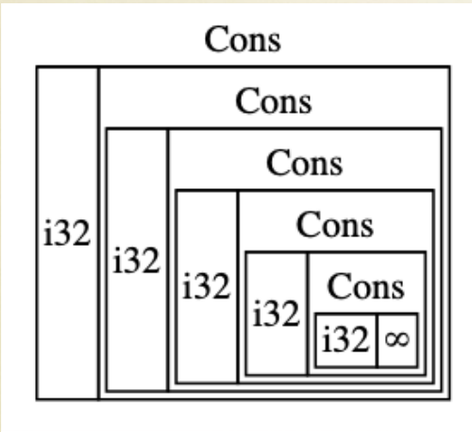
```
enum List { // 크기를 계산할 수 있는가? 스택에 넣으려면 크기를 컴파일 타임에 알아야 함.
    Cons(i32, List),
    Nil,
}
```

```
use List::{Cons, Nil};// 이것을 원하지만….

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

```
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:1:1
  |
1 | enum List {
  | ^^^^^^^^^ recursive type has infinite size
2 |     Cons(i32, List),
  |               ----- recursive without indirection
  |
  = help: insert indirection (e.g., a `Box`, `Rc`, or `&`) at some point to
  make `List` representable
```

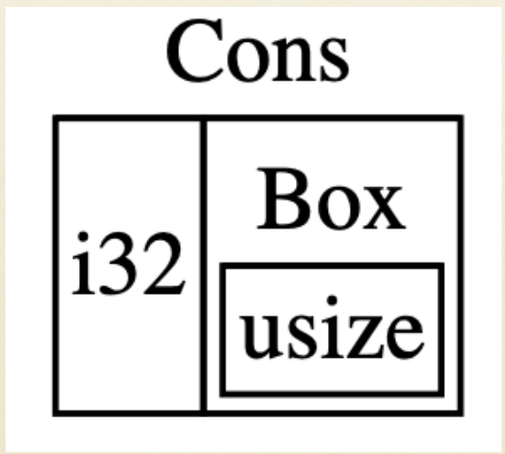
재귀적 타입




```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use List::{Cons, Nil};

fn main() {
    let list = Cons(1,
        Box::new(Cons(2,
            Box::new(Cons(3,
                Box::new(Nil))))));
}
```



Deref trait

- Dereference operator *의 동작을 커스터마이징.

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

    assert_eq!(5, x);
    assert_eq!(5, *y); // 평범한 역참조 연산의 예.
}
```

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y); // Box를 역참조한 예, 정상 동작함.
}
```

커스텀 스마트 포인터

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}

fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}

use std::ops::Deref;
impl<T> Deref for MyBox<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.0
    }
}
```

```
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
  --> src/main.rs:14:19
   |
14 |     assert_eq!(5, *y);
   |                   ^^
```

```
*(y.deref())
```

역참조 강제

- 메소드로 넘기는 인자의 타입이 안 맞을 때 자동으로 호출된다. 즉 역참조를 시행해서 타입을 맞추려 노력한다.

```
fn hello(name: &str) {
    println!("Hello, {}!", name);
}

fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

deref() deref()

&MyBox<String> ==> &String ==> &str

- `T: Deref<Target=U>` 일때 `&T` 에서 `&U` 로
- `T: DerefMut<Target=U>` 일때 `&mut T` 에서 `&mut U` 로
- `T: Deref<Target=U>` 일때 `&mut T` 에서 `&U` 로

가변 참조자는 유일하므로 불변 참조자로 바꾸는게 가능

Drop

- 기본적으로 값이 스코프를 벗어날 때 호출되어 자원을 해제함.
- 이 trait을 구현해서 파일 닫기나 소켓연결 끊기 등을 추가적으로 실행할 수 있음.
- 강제 자원 해제를 하려면 Std::mem::drop 을 사용해야 한다.

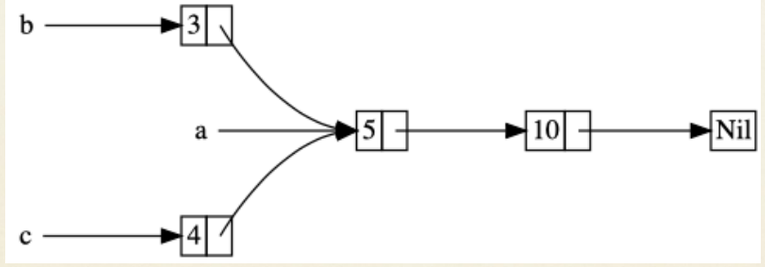
Rc<T>

- 하나의 자원에 여럿이 참조할 때.
- 단일 스레드에서만 사용할 수 있다.

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use List::{Cons, Nil};

fn main() {
    let a = Cons(5,
        Box::new(Cons(10,
            Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```




```
error[E0382]: use of moved value: `a`
 --> src/main.rs:13:30
   |
12 |     let b = Cons(3, Box::new(a));
   |                              - value moved here
13 |     let c = Cons(4, Box::new(a));
   |                              ^ value used here after move
   |
   = note: move occurs because `a` has type `List`, which does not implement
  the `Copy` trait
```

RC 사용 예

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use List::{Cons, Nil};
use std::rc::Rc;

fn main() { // Rc를 사용하여 해결한 예
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

```
fn main() { // 참조 카운터의 변화를 보여주는 예
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
```

```
count after creating a = 1
count after creating b = 2
count after creating c = 3
count after c goes out of scope = 2
```

한계점: 불변 참조만 공유가능하다.

REFCELL<T>

- 내부 가변성(interior mutability): 불변 참조자의 값을 변경가능하도록 해줌
- 일반적 참조자의 검사는 컴파일 타임에 일어남 : 컴파일 에러가 남.
- RefCell의 참조검사는 런 타임에 일어나고, 규칙이 런 타임에 위배되면 panic함.
- 안전하다고 확신할 수 있는 코드는 컴파일을 허용하려는 의도.

요점 정리

- Rc<T>는 동일한 데이터에 대해 복수개의 소유자를 가능하게 합니다; Box<T>와 RefCell<T>은 단일 소유자만 갖습니다.
- Box<T>는 컴파일 타임에 검사된 불변 혹은 가변 빌림을 허용합니다; Rc<T>는 오직 컴파일 타임에 검사된 불변 빌림만 허용합니다; RefCell<T>는 런타임에 검사된 불변 혹은 가변 빌림을 허용합니다.
- RefCell<T>이 런타임에 검사된 가변 빌림을 허용하기 때문에, RefCell<T>이 불변일 때라도 RefCell<T> 내부의 값을 변경할 수 있습니다.

내부가변성의 예

```rust
pub trait Messenger {
    fn send(&self, msg: &str);
}

pub struct LimitTracker<'a, T: 'a + Messenger> {
    messenger: &'a T,
    value: usize,
    max: usize,
}

impl<'a, T> LimitTracker<'a, T>
    where T: Messenger {
    pub fn new(messenger: &T, max: usize) -> LimitTracker<T> {
        LimitTracker {
            messenger,
            value: 0,
            max,
        }
    }

    pub fn set_value(&mut self, value: usize) {
        self.value = value;

        let percentage_of_max = self.value as f64 / self.max as f64;

        if percentage_of_max >= 0.75 && percentage_of_max < 0.9 {
            self.messenger.send("Warning: You've used up over 75% of your quota!");
        } else if percentage_of_max >= 0.9 && percentage_of_max < 1.0 {
            self.messenger.send("Urgent warning: You've used up over 90% of your quota!");
        } else if percentage_of_max >= 1.0 {
            self.messenger.send("Error: You are over your quota!");
        }
    }
}
```

```rust
#[cfg(test)]
mod tests {
    use super::*;

    struct MockMessenger {
        sent_messages: Vec<String>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger { sent_messages: vec![] }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        let mock_messenger = MockMessenger::new();
        let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);

        limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_messages.len(), 1);
    }
}
```

```
error[E0596]: cannot borrow immutable field `self.sent_messages` as mutable
  --> src/lib.rs:52:13
   |
51 |         fn send(&self, message: &str) {
   |                 ----- use `&mut self` here to make mutable
52 |             self.sent_messages.push(String::from(message));
   |             ^^^^^^^^^^^^^^^^^^ cannot mutably borrow immutable field
```

RefCell<T>

```
#[cfg(test)]
mod tests {
    use super::*;
    use std::cell::RefCell;

    struct MockMessenger {
        sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger { sent_messages: RefCell::new(vec![]) }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.borrow_mut().push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        // --snip--

        assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}
```

```
impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        let mut one_borrow = self.sent_messages.borrow_mut();
        let mut two_borrow = self.sent_messages.borrow_mut(); // 여기서 panic

        one_borrow.push(String::from(message));
        two_borrow.push(String::from(message));
    }
}
```

Rc<T>와 RefCell<T>

```
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use List::{Cons, Nil};
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(6)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(10)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a after = {:?}", a);
    println!("b after = {:?}", b);
    println!("c after = {:?}", c);
}
```

```
a after = Cons(RefCell { value: 15 }, Nil)
b after = Cons(RefCell { value: 6 }, Cons(RefCell { value: 15 }, Nil))
c after = Cons(RefCell { value: 10 }, Cons(RefCell { value: 15 }, Nil))
```

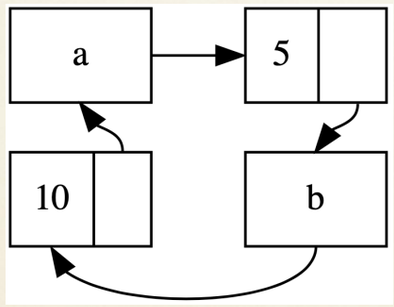
Memory leak

- Rc와 RefCell로 순환 참조를 만들 수 있고 이는 참조 counter가 0이 안되므로, 절대 해제되지 않는 메모리를 가짐 ==> 메모리 누수.

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

#[derive(Debug)]
enum List {
    Cons(i32, RefCell<Rc<List>>),
    Nil,
}

impl List {
    fn tail(&self) -> Option<&RefCell<Rc<List>>> {
        match *self {
            Cons(_, ref item) => Some(item),
            Nil => None,
        }
    }
}
```




```
fn main() {
    let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));

    println!("a initial rc count = {}", Rc::strong_count(&a));
    println!("a next item = {:?}", a.tail());

    let b = Rc::new(Cons(10, RefCell::new(Rc::clone(&a))));

    println!("a rc count after b creation = {}", Rc::strong_count(&a));
    println!("b initial rc count = {}", Rc::strong_count(&b));
    println!("b next item = {:?}", b.tail());

    if let Some(link) = a.tail() {
        *link.borrow_mut() = Rc::clone(&b);
    }

    println!("b rc count after changing a = {}", Rc::strong_count(&b));
    println!("a rc count after changing a = {}", Rc::strong_count(&a));

    // Uncomment the next line to see that we have a cycle;
    // it will overflow the stack
    // println!("a next item = {:?}", a.tail());
}
```

참조 순환 방지: Weak

- strong_count, weak_count
- Rc::downgrade -> Weak<T>
- Weak::upgrade -> Option<Rc<T>>
- Weak가 가리키는 값을 접근할 때 반드시 유효한지 확인. ==> upgrade 호출하여 None이면 해제된 자원임.

트리

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

#[derive(Debug)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());

    let branch = Rc::new(Node {
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch);

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

Q & A