APPS@UCU

Rust #4: Smart pointers, Concurrency and Async

Sultanov Andriy

Contents

Smart pointers

Smart pointers
General

Rust standard library has a few useful smart pointers, which modify the possibilities of operating with the value they hold:

- Box<T>
- Rc<T>
- Cell<T>
- RefCell<T>

Smart pointers
Deref trait

Implementing Deref allows Rust to treat smart pointers as references to the values they hold.

You can dereference types impementing Deref explicitly using the * operator, or you can omit the dereferencing and let Rust implicitly coerce your type to the type you need.

Thus, when calling a method on a type that implements Deref and does not implement the method, Rust will recursively dereference this type until it finds the appropriate method.

Smart pointers
Deref trait

So, let's say we want to have this kind of OOP behavior in Rust:

```
class Foo {
    void m() { ... }
}

class Bar extends Foo {}

public static void main(String[] args) {
    Bar b = new Bar();
    b.m();
}
```

Smart pointers
Deref trait

We can model it like this:

```
use std::ops::Deref;

struct Foo {}

impl Foo {
    fn m(&self) {
        //..
    }
}

struct Bar {
    f: Foo,
}

impl Deref for Bar {
    type Target = Foo;
    fn deref(&self) -> &Foo {
        &self.f
    }
}
```

And then easily use it:

```
fn main() {
    let b = Bar { f: Foo {} };
    b.m();
}
```

This allows you to easily operate with Rust's smart pointers.

Smart pointers
Deref trait

Rust will coerce types according to these rules:

- From &T to &U when T: Deref<Target=U>
- From &mut T to &mut U when T: DerefMut<Target=U>
- From &mut T to &U when T: Deref<Target=U>

Box<T>

Boxes are Rust’s way to store data on the heap.

They are useful when you want to handle values whose sizes can not be known at compile time as if their size was known. This works since the size of the pointer to the heap that the Box keeps on the stack is known.

Box<T>

An example of moving a type that is typically stored on the stack onto the heap:

```
let val: u8 = 42;
let boxed: Box<u8> = Box::new(val);

// We can easily dereference the box
println!("{}", boxed); // prints 42
```

The value is going to be deallocated once it goes out scope, calling Box's drop implementation.

Box<T>

We could have just as well explicitly dereferenced the Box:

```
let val: u8 = 42;
let boxed: Box<u8> = Box::new(val);

// We can easily dereference the box
println!("{}", boxed); // prints 42

// And explicitly like this:
println!("{}", *boxed);
```

Box<T>

If you want to create an unsized data structure, for example a recursive cons list, you can just use Box:

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

// Creating a new list
let list: List<i32> = List::Cons(1, Box::new(List::Cons(2, Box::new(List::Nil))));
println!("{:?}", list); // prints Cons(1, Cons(2, Nil))
```

What will end up being stored is an i32 and a pointer usize to another List on the heap.

Rc<T>

While Rust's ownership system is pretty strict, there are some ways to have multiple ownership. Rc<T> is a reference counter that keeps track of the number of pointers to a certain value, and drops it once nobody uses it anymore.

Rc<T>

Rc<T> does not allow mutability, and basically changes the ownership system so that the control over the lifetime of the value is done during runtime, not compile time.

Rc<T>

```
let text = "Rc examples".to_string();
{
    // Reference count is 1
    let rc_a: Rc<String> = Rc::new(text);
    {
        // Reference count is 2
        let rc_b: Rc<String> = Rc::clone(&rc_a);

        rc_a.len();
        println!("{}", rc_b);

        // rc_b is dropped, reference count is 1
    }
    // rc_a is dropped, reference count is 0, the value is dropped too
}
// Error, rc_examples was dropped!
println!("rc_examples: {}", rc_examples);
```

Cell<T>

TODO

RefCell<T>

TODO

Concurrency

Concurrency

TODO: Talk about Arc, Mutex, mspc, and the way Rust's system prevents data races at compile time.

Looking back

Looking back

Over the course of four weeks we've covered some general topics, which I hope you will be able to apply everywhere, not only in Rust:

- Machine memory model
- Safe mutability rules
- Safe ownership and lifetime rules
- Multithreading

Looking back

And we've also looked at quite a lot of Rust-specific stuff:

- Rust's ownership/lifetime/mutability system
- Cargo ecosystem
- Algebraic types and error handling
- Declarative macros
- Traits
- Iterators and closures
- Multithreading primitives

And ahead...

Looking ahead

There is quite a lot of stuff in Rust that we haven't covered. If you want to dive deeper yourself, here is a short list of some of the most important things you can look into:

- Async Rust
- Procedural macros
- Testing
- FFI and bindings

And many more...

Looking ahead

But, most importantly, practice makes perfect!

I hope that this short tour of Rust has shown you the possibilities of modern languages and ecosystems, and taught you a few important things in general!

Thank you!