2_Smart pointers_en

May 2, 2023

1 Smart pointers

A pointer is a general concept for a variable that contains an address in memory. This address refers to, or "points at," some other data. The most common kind of pointer in Rust is a *reference*.

Smart pointers, on the other hand, are data structures that not only act like a pointer but also have additional metadata and capabilities. References are pointers that only borrow data; in contrast, in many cases, smart pointers own the data they point to. Strings and vectors are exmples of smart pointers.

Smart pointers are usually implemented using structs. The characteristic that distinguishes a smart pointer from an ordinary struct is that smart pointers implement the **`Deref`** and **`Drop`** traits.

Many libraries have their own smart pointers, and you can even write your own. The most common smart pointers in the standard library are:

- `Box<T>` for allocating values on the heap
- `Rc<T>`, a reference counting type that enables multiple ownership
- `Ref<T>` and `RefMut<T>`, accessed through `RefCell<T>`, a type that enforces the borrowing rules at runtime instead of compile time

1.1 Box<T>

Boxes allow you to store data on the heap rather than the stack. What remains on the stack is the pointer to the heap data.

You'll use them most often in these situations:

- When you have a type whose size can't be known at compile time and you want to use a value of that type in a context that requires an exact size
- When you have a large amount of data and you want to transfer ownership but ensure the data won't be copied when you do so
- When you want to own a value and you care only that it's a type that implements a particular trait rather than being of a specific type

[ ]:
```rust
fn main() {
    let b = Box::new(5); // store an i32 value on the heap
    println!("b = {}", b);
} // box is deallocated here, because it becomes out of scope
```

At compile time, Rust needs to know how much space a type takes up. One type whose size can't be known at compile time is a recursive type, where a value can have as part of itself another value of the same type. Because this nesting of values could theoretically continue infinitely, Rust doesn't know how much space a value of a recursive type needs. However, boxes have a known size, so by inserting a box in a recursive type definition, you can have recursive types.

The *cons list* data type

Each item in a **cons list** contains two elements: the value of the current item and the next item. The last item in the list contains only a value called Nil without a next item. A cons list is produced by recursively calling the cons function.

```
// Wrong example of a recursive List definition because the size would be infinite
enum List {
    Cons(i32, List),
    Nil,
}
```

[ ]:
```
enum List {
    Cons(i32, Box<List>), // the size of an i32 plus the space to store the␣
 ↪box's pointer data
    Nil,
}

use crate::List::{Cons, Nil};

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

Treating smart pointers like regular references with the Deref trait

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

    println!("x = {}", x);
    println!("y points to the value: {}", *y);
}
main();
```

[ ]:
```
fn main() {
    let x = 5;
    let y = Box::new(x); // using Box <T> like a reference

    println!("x = {}", x);
    println!("y points to the value: {}", *y); // it works because Box␣
 ↪implements the Deref trait
}
main();
```

1.2 Define your smart pointer

[ ]:
```
use std::ops::Deref;

fn main() {
    // The MyBox type is a tuple struct with one element of type T
    struct MyBox<T>(T); // can hold values of any type

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

    impl<T> Deref for MyBox<T> {
        type Target = T;  // defines an associated type for the Deref trait to␣
 ↪use

        fn deref(&self) -> &T {
            &self.0
        }
    }
    let x = 5;
    let y = MyBox::new(x);

    println!("x = {}", x);
    println!("y points to the value: {}", *y); // Rust runs the code: *(y.
 ↪deref())
}
main();
```

Without the `Deref` trait, the compiler can only dereference `&` references. The `deref` method gives the compiler the ability to take a value of any type that implements `Deref` and call the `deref` method to get a `&` reference that it knows how to dereference.

Implicit Deref coercions with functions and methods

*`Deref` **coercion** is a convenience that Rust performs on arguments to functions and methods. `Deref` coercion converts a reference to a type that implements `Deref` into a reference to a type that `Deref` can convert the original type into. `Deref` coercion happens automatically when we pass a reference to a particular type's value as an argument to a function or method that doesn't match the parameter type in the function or method definition. A sequence of calls to the `deref` method converts the type we provided into the type the parameter needs.

[ ]:
```rust
use std::ops::Deref;

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

fn main() {
    struct MyBox<T>(T);

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

    impl<T> Deref for MyBox<T> {
        type Target = T;

        fn deref(&self) -> &T {
            &self.0
        }
    }
    let m = MyBox::new(String::from("Rust"));
    hello(&m); // otherwise, we should write: hello(&(*m)[..]);
}
main();
```

Here we're calling the `hello` function with the argument `&m`, which is a reference to a `MyBox<String>` value. Because we implemented the `Deref` trait on `MyBox<T>`, Rust can turn `&MyBox<String>` into `&String` by calling `deref`. The standard library provides an implementation of `Deref` on `String` that returns a string slice. Rust calls `deref` again to turn the `&String` into `&str`, which matches the `hello` function's definition.

Rust does deref coercion when it finds types and trait implementations in three cases:

- 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>`

Rust will also coerce a mutable reference to an immutable one. But the reverse is not possible: immutable references will never coerce to mutable references.

[ ]:
```rust
use std::ops::Deref;
use std::fmt::Debug;

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

```

```
fn main() {
    struct MyBox<T: Debug>(T);

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

    impl<T> Deref for MyBox<T>
    where T: Debug {
        type Target = T;

        fn deref(&self) -> &T {
            &self.0
        }
    }

    impl<T> Drop for MyBox<T>
    where T: Debug {
        fn drop(&mut self) {
            println!("Dropping pointer with data `{:?}`!", self.0);
        }
    }
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
main();
```

1.3 `Rc<T>`, the reference counted smart pointer

In the majority of cases, ownership is clear: you know exactly which variable owns a given value. However, there are cases when a single value might have multiple owners. For example, in graph data structures, multiple edges might point to the same node, and that node is conceptually owned by all of the edges that point to it. A node shouldn't be cleaned up unless it doesn't have any edges pointing to it.

To enable multiple ownership, Rust has a type called `Rc<T>`, which is an abbreviation for *reference counting*. The `Rc<T>` type keeps track of the number of references to a value which determines whether or not a value is still in use. If there are zero references to a value, the value can be cleaned up without any references becoming invalid.

We use the **`Rc<T>`** type when we want to allocate some data on the heap for multiple parts of our program to read and we can't determine at compile time which part will finish using the data last. Note that `Rc<T>` is only for use in single-threaded scenarios.

Consider the example bellow:

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

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

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    // increase the number of references from one to two and let a and b share␣
 ↪ownership of the data in that Rc<List>
    let b = Cons(3, Rc::clone(&a));
    // increase the number of references to three
    let c = Cons(4, Rc::clone(&a));
}
main();
```

When we create b, instead of taking ownership of a, we'll clone the `Rc<List>` that a is holding, thereby increasing the number of references from one to two and letting a and b share ownership of the data in that `Rc<List>`. We'll also clone a when creating c, increasing the number of references from two to three. Every time we call `Rc::clone`, the reference count to the data within the `Rc<List>` will increase, and the data won't be cleaned up unless there are zero references to it.

In the following example, we can see the reference counts changing as we create and drop references to the Rc in a:

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

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

fn main() {
    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));
    {   // inner scope around list c
        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));
}
main();
```

1.4 RefCell<T> and the Interior Mutability Pattern

Interior mutability is a design pattern in Rust that allows you to *mutate data even when there are immutable references to that data*; normally, this action is disallowed by the borrowing rules.

With references and Box<T>, the borrowing rules' invariants are enforced at *compile time*. With RefCell<T>, these invariants are enforced at *runtime*. The RefCell<T> type is useful when you're sure your code follows the borrowing rules but the compiler is unable to understand and guarantee that. RefCell<T> is only for use in single-threaded scenarios.

[ ]:
```
use std::cell::RefCell;
fn main() {
   let c = RefCell::new(5);
   let m = c.borrow();
   println!("{}", *c.borrow());
   // multiple immutable borrows are allowed
   // a reference count is increased at each immutable borrow
   let p = c.borrow();
}
main();
```

[ ]:
```
use std::cell::RefCell;
fn main() {
   let c = RefCell::new(5);
   // the reference count is set to 0 when mutably borrowed
   let k = c.borrow_mut();
   // only one mutable borrow allowed at any point in time
   // next line would cause a panic
   let l = c.borrow();
}
main();
```

[ ]:
```
fn main() {
    let x = RefCell::new(vec![1,2,3,4]);

    let v = x.borrow();
    println!("{:?}", *v);

    let mut my_ref = x.borrow_mut(); // cannot borrow as mutable, the thread␣
 ↪panics
    my_ref.push(1);
}
main();
```

With RefCell<T>, we use the **borrow** and **borrow_mut** methods, which are part of the safe API that belongs to RefCell<T>. The borrow method returns the smart pointer type **Ref<T>**, and borrow_mut returns the smart pointer type **RefMut<T>**. Both types implement Deref, so we can treat them like regular references.

The `RefCell<T>` keeps track of how many `Ref<T>` and `RefMut<T>` smart pointers are currently active. Every time we call `borrow`, the `RefCell<T>` increases its count of how many immutable borrows are active. When a `Ref<T>` value goes out of scope, the count of immutable borrows goes down by one. Just like the compile-time borrowing rules, `RefCell<T>` lets us have many immutable borrows or one mutable borrow at any point in time.

Exercise 1: Which statement applies to which data type (Box, Rc, RefCell) ? * enables multiple owners of the same data * has single owner * allows immutable borrows checked at compile time * allows mutable borrows checked at compile time * allows immutable borrows checked at runtime * allows mutable borrows checked at runtime

Solution:

Here is a recap of the reasons to choose `Box<T>`, `Rc<T>`, or `RefCell<T>`:

- `Rc<T>` enables multiple owners of the same data; `Box<T>` and `RefCell<T>` have single owners.
- `Box<T>` allows immutable or mutable borrows checked at compile time; `Rc<T>` allows only immutable borrows checked at compile time; `RefCell<T>` allows immutable or mutable borrows checked at runtime.
- Because `RefCell<T>` allows mutable borrows checked at runtime, you can mutate the value inside the `RefCell<T>` even when the `RefCell<T>` is immutable.