4_Closures_en

May 2, 2023

1 Closures: Anonymous Functions that Can Capture Their Environment

Rust's closures are anonymous functions you can save in a variable or pass as arguments to other functions. You can create the closure in one place and then call the closure to evaluate it in a different context.

The syntax and capabilities of closures make them very convenient for on the fly usage. Calling a closure is exactly like calling a function. However, both input and return types can be inferred and input variable names must be specified.

Other characteristics of closures include:

- using `| |` instead of `()` around input variables.
- optional body delimination (`{}`) for a single expression (mandatory otherwise).
- the ability to capture the outer environment variables.

[ ]:
```rust
fn main() {
    // Increment via closures and functions.
    fn  function (i: i32) -> i32 { i + 1 }

    // Closures are anonymous, here we are binding them to references
    // Annotation is identical to function annotation but is optional
    // as are the `{}` wrapping the body. These nameless functions
    // are assigned to appropriately named variables.
    let closure_annotated = |i: i32| -> i32 { i + 1 };
    let closure_inferred  = |i     |          i + 1  ;

    let i = 1;
    println!("function: {}", function(i));
    println!("closure_annotated: {}", closure_annotated(i));
    println!("closure_inferred: {}", closure_inferred(i));

}
main();
```

Unlike functions, closures can capture values from the scope in which they're defined.

[ ]:
```
fn main() {
    let value = 3;

    let f = |x| x + value; // the closure here refers to value
    println!("{}", f(2));
}
main();
```

By default, closures capture values as (immutable) references :

[ ]:
```
fn main() {
    let value = String::from("Hello !");
    let tell = || { println!("{}", value); };
    tell();
    println!("value is still here: {}", value);
}
main();
```

Closures can also capture mutable references, by marking the variable that owns the closure as mutable :

[ ]:
```
fn main() {
    let mut value = 0;
    {
        let mut inc = || { value += 1; };
        inc();
    }
    println!("value = {}", value);
}
main();
```

The borrow-checker ensures the safety of the mutable references even for closures. In the code above, remove the inner block to put all statements into the block of the `main` function and see what happens.

Sometimes it is useful for the closure to take ownership of its environment. For this purpose, use the `move` keyword in the closure definition. This will mean that the closure takes ownership and moves all the data it refers to :

[ ]:
```
fn main() {
    let a = String::from("Hello ");
    let b = String::from("World !");
    let tell = move || { print!("{}", a); }; // here a is moved to the closure

    tell();
    println!("{}", b); // this is OK
    // println!("{}", a); // this is an error
}
main();
```

Closures don't require you to annotate the types of the parameters or the return value like fn functions do. Closure definitions will have one concrete type inferred for each of their parameters and for their return value.

[ ]:
```
fn generate_workout(intensity: u32, random_number: u32) {
    let expensive_closure = |num| {
        println!("calculating slowly...");
        // do something here that takes time to execute
        num
    };

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_closure(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_closure(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_closure(intensity)
            );
        }
    }
}
```

1.1 Storing closures using generic parameters and the `Fn` traits

Notice that in the previous code, the expensive_closure, will be executed more times than needed, which takes even more time.

To avoid this, we can create a struct that will hold the closure and the resulting value of calling the closure. The struct will execute the closure only if we need the resulting value, and it will cache the resulting value so the rest of our code doesn't have to be responsible for saving and reusing the result. You may know this pattern as **memoization** or **lazy evaluation**.

To define structs, enums, or function parameters that use closures, we use generics and trait bounds.

All closures implement at least one of the traits: **`Fn`**, **`FnMut`**, or **`FnOnce`**. The **`Fn`** traits are provided by the standard library.

[ ]:
```
// the Cacher struct holds a closure and an optional result value
struct Cacher<T>
```

```
    where T: Fn(u32) -> u32
{
    calculation: T,
    value: Option<u32>,
}

impl<T> Cacher<T>
    where T: Fn(u32) -> u32
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            },
        }
    }
}
```

The `Cacher` struct has a calculation field of the generic type `T`. The trait bounds on `T` specify that it's a closure by using the `Fn` trait. Any closure we want to store in the calculation field must have one `u32` parameter (specified within the parentheses after `Fn`) and must return a `u32` (specified after the `->`).

Before we execute the closure, value will be `None`. When code using a `Cacher` asks for the result of the closure, the `Cacher` will execute the closure at that time and store the result within a `Some` variant in the value field. Then if the code asks for the result of the closure again, instead of executing the closure again, the `Cacher` will return the result held in the `Some` variant.

> **Note:** Functions can implement all three of the `Fn` traits too. If what we want to do doesn't require capturing a value from the environment, we can use a function rather than a closure where we need something that implements an `Fn` trait.

[ ]:
```
struct Cacher<T>
    where T: Fn(u32) -> u32
{
    calculation: T,
    value: Option<u32>,
}
```

```rust
impl<T> Cacher<T>
    where T: Fn(u32) -> u32
{
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            },
        }
    }
}

fn generate_workout(intensity: u32, random_number: u32) {
    // using the Cacher struct
    let mut expensive_result = Cacher::new(|num| {
        println!("calculating slowly...");
        // do something here that takes time
        num
    });

    if intensity < 25 {
        println!(
            "Today, do {} pushups!",
            expensive_result.value(intensity)
        );
        println!(
            "Next, do {} situps!",
            expensive_result.value(intensity)
        );
    } else {
        if random_number == 3 {
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes!",
                expensive_result.value(intensity)
```

```
            );
        }
    }
}

fn main()
{
    let mut c = Cacher::new(|a| a);

    let v1 = c.value(1);
    generate_workout(v1, 7);

}
main();
```

Instead of saving the closure in a variable directly, we save a new instance of `Cacher` that holds the closure. Then, in each place we want the result, we call the value method on the `Cacher` instance. We can call the value method as many times as we want, or not call it at all, and the expensive calculation will be run a maximum of once.

There are some limitations of this implementation: it assumes it will always get the same value for the parameter `arg` to the value method, and it only accepts closures that take one parameter of type `u32` and return a `u32`. To solve this issue, generic parameters need to be used.

1.2 Capturing the environment with closures

Closures can capture values from their environment in three ways, which directly map to the three ways a function can take a parameter: taking ownership, borrowing mutably, and borrowing immutably. These are encoded in the three `Fn` traits as follows:

- `FnOnce` consumes the variables it captures from its enclosing scope, known as the *closure's environment*. To consume the captured variables, the closure must take ownership of these variables and move them into the closure when it is defined. The Once part of the name represents the fact that the closure can't take ownership of the same variables more than once, so it can be called only once.
- `FnMut` can change the environment because it mutably borrows values.
- `Fn` borrows values from the environment immutably.