Fn Traits

Florian "Florob" Zeitz

2018-09-05

Greeter


```
1  struct Greeter;
2
3  impl Greeter {
4    fn call(...) {
5      println!("Hello Rust");
6    }
7  }
```

What is the most logical choice for `call`s argument?

- A `self`
- B `&self`
- C `&mut self`
- D Nothing (not a method)

Greeter


```
1  struct Greeter;
2
3  impl Greeter {
4    fn call(...) {
5      println!("Hello Rust");
6    }
7  }
```

What is the most logical choice for `call`s argument?

- A `self`
- B `&self`
- C `&mut self`
- D Nothing (not a method)

Greeter


```
1  struct Greeter;
2
3  impl Greeter {
4      fn call(&self) {
5          println!("Hello Rust");
6      }
7
8      fn call_mut(&mut self) {
9          self.call()
10     }
11
12     fn call_once(self) {
13         self.call()
14     }
15 }
```

- We can implement all variants
- `&mut self` can be re-borrowed as `&self`
- `self` can be borrowed as `&self`

More Useful Greeter


```
1  struct Greeter(String);
2
3  impl Greeter {
4      fn call(&self) {
5          println!("Hello {}", self.0);
6      }
7
8      fn call_mut(&mut self) {
9          self.call()
10     }
11
12     fn call_once(self) {
13         self.call()
14     }
15 }
```

- What if we attach data to our `struct`?
- Makes no difference as long as we only use it by `&T` reference

Fibonacci


```
1  struct Fib(u64, u64);
2
3  impl Fib {
4    fn call(...) -> u64 {
5      let res = self.0;
6      self.0 = self.1;
7      self.1 += res;
8      res
9    }
10 }
```

What is the most logical choice for `call`s argument?

A. `self`
B. `&self`
C. `&mut self`
D. Nothing (not a method)

Fibonacci


```
1  struct Fib(u64, u64);
2
3  impl Fib {
4    fn call(...) -> u64 {
5      let res = self.0;
6      self.0 = self.1;
7      self.1 += res;
8      res
9    }
10 }
```

What is the most logical choice for `call`s argument?

- A `self`
- B `&self`
- C `&mut self`
- D Nothing (not a method)

Fibonacci


```
1  struct Fib(u64, u64);
2
3  impl Fib {
4      fn call_mut(&mut self) -> u64 {
5          let res = self.0;
6          self.0 = self.1;
7          self.1 += res;
8          res
9      }
10
11     fn call_once(mut self) -> u64 {
12         self.call_mut()
13     }
14 }
```

- Obviously can't do `&self` anymore
- `self` can be borrowed as `&mut self`

Nonce


```
1  struct Nonce(Vec<u8>);
2
3  impl Fib {
4    fn call(...) -> Vec<u8> {
5      self.0
6    }
7  }
```

What is the most logical choice for `call`s argument?

A `self`

B `&self`

C `&mut self`

D Nothing (not a method)

Nonce


```
1  struct Nonce(Vec<u8>);
2
3  impl Fib {
4    fn call(...) -> Vec<u8> {
5      self.0
6    }
7  }
```

What is the most logical choice for `call`s argument?

A `self`

B `&self`

C `&mut self`

D Nothing (not a method)

Nonce


```
1  struct Nonce(Vec<u8>);
2
3  impl Fib {
4    fn call_once(self) -> Vec<u8> {
5      self.0
6    }
7  }
```

- Has to take `self`
- Moves its inner value out once

Some Traits

We've seen there is no one size fits all solution. We can define some traits though:

```
1  trait FnOnce {
2    type Output;
3    fn call_once(self) -> Self::Output;
4  }
5
6  trait FnMut: FnOnce {
7    fn call_mut(&mut self) -> Self::Output;
8  }
9
10 trait Fn: FnMut {
11   fn call(&self) -> Self::Output;
12 }
```

Arguments

- Aren't we missing something?
- With our current traits we can return any Type, but take no arguments
- We want to take an arbitrary number of parameters with varying types (variadic)
- Rust has no variadic generics

The Real Traits


```
1  trait FnOnce<Args> {
2    type Output;
3    extern "rust-call" fn call_once(self, args: Args) ->
     ↪  Self::Output;
4  }
5
6  trait FnMut<Args>: FnOnce<Args> {
7    extern "rust-call" fn call_mut(&mut self, args: Args) ->
     ↪  Self::Output;
8  }
9
10 trait Fn<Args>: FnMut<Args> {
11   extern "rust-call" fn call(&self, args: Args) -> Self::Output;
12 }
```

These traits are all unstable to implement.

The Catch

- We can't `impl` these traits ourselves
- Closures automatically implement them
- Functions automatically implement them
- We can't directly use these traits as bounds
- There is sugar for `Fn` trait bounds

Greeter


```
1  let mut greeter =
2    || println!("Hello Rust!");
3
4  greeter();
5  greeter.call(());
6  greeter.call_mut(());
7  greeter.call_once(());
```

- Desugars to the same thing as our first example
- Except it implements the `Fn` traits
- All possible ones

More Useful Greeter


```
1  let name = "Cologne".to_string();
2  let mut greeter =
3    || println!("Hello {}!", name);
4
5  greeter();
6  greeter.call(());
7  greeter.call_mut(());
8  greeter.call_once(());
```

- This is the same as our second example.
- Or is it?

More Useful Greeter


```
1  fn greeter(
2    name: String
3  ) -> impl Fn() {
4    || println!("Hello {}!", name)
5  }
6
7  let mut greeter =
8    greeter("Cologne".into());
9
10 greeter();
11 greeter.call(());
12 greeter.call_mut(());
13 greeter.call_once(());
```

- closure may outlive the current function, but it borrows `name`, which is owned by the current function
- Wait... borrows?

More Useful Greeter


```
1  struct Greeter {
2    name: &String
3  }
4
5  // Pseudocode!
6  impl Fn for Greeter {
7    fn call(&mut self) {
8      println!("Hello {}!", self.name);
9    }
10 }
```

- This is the actual desugaring
- Closures capture variables based on their usage
- Capturing means taking a reference, or storing (moving) the value

More Useful Greeter


```
1  fn greeter(
2    name: String
3  ) -> impl Fn() {
4    move || println!("Hello
     ↪  {}!", name)
5  }
6
7  let mut greeter =
8    greeter("Cologne".into());
9
10 greeter();
11 greeter.call(());
12 greeter.call_mut(());
13 greeter.call_once(());
```

- Actually desugars to our second example
- `move` enforces moving the values captured from the environment
- Closures can still contain references, in case the captured value is a reference

Fibonacci


```
1  let mut a = 1;
2  let mut b = 1;
3  let mut fib = move || {
4      let res = a;
5      a = b;
6      b += res;
7      res
8  };
9
10 fib();
11 // error: `Fn` is not implemented
12 // fib.call(());
13 fib.call_mut(());
14 fib.call_once(());
```

- Desugars to our Fibonacci example
- Implementing `Fn` is not possible, so the compiler doesn't

Nonce


```
1  let n = vec![4, 8, 15, 16, 23, 42];
2  let nonce = || n;
3
4  nonce();
5  // error: `Fn` is not implemented
6  // nonce.call(());
7  // error: `FnMut` is not implemented
8  // nonce.call_mut(());
9  // error: value has been moved ;)
10 nonce.call_once(());
```

- Desugars to our Nonce example
- Implementing `Fn` and `FnMut` is not possible, so the compiler doesn't
- Behaves the same with and without **`move`**, closure has to move the value to return it

Fn Trait Bounds

- Can't use the unstable `Fn` traits as bounds directly
- But there is sugar that we can use
- And it conveniently looks like a function signature
  - `Fn(u32) -> u64`
  - `FnMut(&str) -> i32`
  - `FnOnce(u8) -> String`

Map One


```
1  struct Abstraction<T>(T);
2
3  impl<T> Abstraction<T> {
4    fn map<U, F>(self, f: F) -> Abstraction<U>
5      where F: FnOnce(T) -> U
6    {
7      Abstraction(f(self.0))
8    }
9  }
```

While it may seem strange at first `FnOnce` is the most general trait bound. Everything can be called at least once.

Map Two


```
1  struct Abstraction2<T>(T, T);
2
3  impl<T> Abstraction2<T> {
4    fn map<U, F>(self, mut f: F) -> Abstraction2<U>
5      where F: FnMut(T) -> U
6    {
7      Abstraction2(f(self.0), f(self.1))
8    }
9  }
```

When you need to call the function more than once `FnMut` is the next best option. Choosing `Fn` as a bound is rare as it gives the calling function almost no additional options.

Give me five


```
1  fn give5() -> impl Fn() -> u64 {
2      || 5
3  }
4
5  let dave_brubeck = give5();
6  assert_eq!(5, dave_brubeck());
```

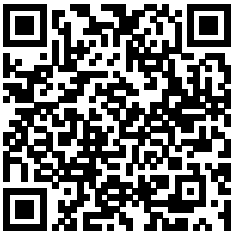
When returning a closure `Fn` is most general since it allows calling the object multiple times and behind any kind of reference.

Give me 5n


```
1  fn counter() -> impl FnMut() -> u64 {
2      let mut n = 0;
3      move || {
4          n += 1;
5          5 * n
6      }
7  }
```

Depending on the traits the closure can actually implement, we fall back to `FnMut` or `FnOnce`.

Thank you for your attention. Any questions?



https://babelmonkeys.de/~florob/talks/RC-2018-09-05-fn-traits.pdf