闭包 Closure

闭包这个词语由来已久，自上世纪 60 年代就由 `Scheme` 语言引进之后，被广泛用于函数式编程语言中，进入 21 世纪后，各种现代化的编程语言也都不约而同地把闭包作为核心特性纳入到语言设计中来。那么到底何为闭包?

闭包是**一种匿名函数，它可以赋值给变量也可以作为参数传递给其它函数，不同于函数的是，它允许捕获调用者作用域中的值**，例如：

```
fn main() {
   let x = 1;
   let sum = |y| x + y;

    assert_eq!(3, sum(2));
}
```

上面的代码展示了非常简单的闭包 `sum`，它拥有一个入参 `y`，同时捕获了作用域中的 `x` 的值，因此调用 `sum(2)` 意味着将 2（参数 `y`）跟 1（`x`）进行相加，最终返回它们的和：`3`。

可以看到 `sum` 非常符合闭包的定义：可以赋值给变量，允许捕获调用者作用域中的值。

使用闭包来简化代码

传统函数实现

想象一下，我们要进行健身，用代码怎么实现（写代码什么鬼，健身难道不应该去健身房嘛？答曰：健身太累了，还是虚拟健身好，点到为止）？这里是我的想法：

```rust
use std::thread;
use std::time::Duration;

// 开始健身，好累，我得发出声音：muuuu...
fn muuuuu(intensity: u32) -> u32 {
    println!("muuuu.....");
    thread::sleep(Duration::from_secs(2));
    intensity
}

fn workout(intensity: u32, random_number: u32) {
    if intensity < 25 {
        println!(
            "今天活力满满，先做 {} 个俯卧撑!",
            muuuuu(intensity)
        );
        println!(
            "旁边有妹子在看，俯卧撑太low，再来 {} 组卧推!",
            muuuuu(intensity)
        );
    } else if random_number == 3 {
        println!("昨天练过度了，今天还是休息下吧！");
    } else {
        println!(
            "昨天练过度了，今天干干有氧，跑步 {} 分钟!",
            muuuuu(intensity)
        );
    }
}

fn main() {
    // 强度
    let intensity = 10;
    // 随机值用来决定某个选择
    let random_number = 7;

    // 开始健身
    workout(intensity, random_number);
}
```

可以看到，在健身时我们根据想要的强度来调整具体的动作，然后调用 `muuuuu` 函数来开始健身。这个程序本身很简单，没啥好说的，但是假如未来不用 `muuuuu` 函数了，是不是得把所有 `muuuuu` 都替换成，比如说 `woooo` ？如果 `muuuuu` 出现了几十次，那意味着我们要修改几十处地方。

函数变量实现

一个可行的办法是，把函数赋值给一个变量，然后通过变量调用：

```rust
use std::thread;
use std::time::Duration;

// 开始健身，好累，我得发出声音：muuuu...
fn muuuuu(intensity: u32) -> u32 {
    println!("muuuu.....");
    thread::sleep(Duration::from_secs(2));
    intensity
}

fn workout(intensity: u32, random_number: u32) {
    let action = muuuuu;
    if intensity < 25 {
        println!(
            "今天活力满满，先做 {} 个俯卧撑!",
            action(intensity)
        );
        println!(
            "旁边有妹子在看，俯卧撑太low，再来 {} 组卧推!",
            action(intensity)
        );
    } else if random_number == 3 {
        println!("昨天练过度了，今天还是休息下吧！");
    } else {
        println!(
            "昨天练过度了，今天干干有氧，跑步 {} 分钟!",
            action(intensity)
        );
    }
}

fn main() {
    // 强度
    let intensity = 10;
    // 随机值用来决定某个选择
    let random_number = 7;

    // 开始健身
    workout(intensity, random_number);
}
```

经过上面修改后，所有的调用都通过 `action` 来完成，若未来声(动)音(作)变了，只要修改为 `let action = woooo` 即可。

但是问题又来了，若 `intensity` 也变了怎么办？例如变成 `action(intensity + 1)`，那你又得哐哐哐修改几十处调用。

该怎么办？没太好的办法了，只能祭出大杀器：闭包。

闭包实现

上面提到 `intensity` 要是变化怎么办，简单，使用闭包来捕获它，这是我们的拿手好戏：

```rust
use std::thread;
use std::time::Duration;

fn workout(intensity: u32, random_number: u32) {
    let action = || {
        println!("muuuu.....");
        thread::sleep(Duration::from_secs(2));
        intensity
    };

    if intensity < 25 {
        println!(
            "今天活力满满，先做 {} 个俯卧撑!",
            action()
        );
        println!(
            "旁边有妹子在看，俯卧撑太low，再来 {} 组卧推!",
            action()
        );
    } else if random_number == 3 {
        println!("昨天练过度了，今天还是休息下吧！");
    } else {
        println!(
            "昨天练过度了，今天干干有氧，跑步 {} 分钟!",
            action()
        );
    }
}

fn main() {
    // 动作次数
    let intensity = 10;
    // 随机值用来决定某个选择
    let random_number = 7;

    // 开始健身
    workout(intensity, random_number);
}
```

在上面代码中，无论你要修改什么，只要修改闭包 `action` 的实现即可，其它地方只负责调用，完美解决了我们的问题！

Rust 闭包在形式上借鉴了 `Smalltalk` 和 `Ruby` 语言，与函数最大的不同就是它的参数是通过 `|parm1|` 的形式进行声明，如果是多个参数就 `|param1, param2,...|`， 下面给出闭包的形式定义：

```
|param1, param2,...| {
    语句1;
    语句2;
    返回表达式
}
```

如果只有一个返回表达式的话，定义可以简化为：

```
|param1| 返回表达式
```

上例中还有两点值得注意：

- **闭包中最后一行表达式返回的值，就是闭包执行后的返回值**，因此 `action()` 调用返回了 `intensity` 的值 `10`
- `let action = ||...` 只是把闭包赋值给变量 `action`，并不是把闭包执行后的结果赋值给 `action`，因此这里 `action` 就相当于闭包函数，可以跟函数一样进行调用： `action()`

闭包的类型推导

Rust 是静态语言，因此所有的变量都具有类型，但是得益于编译器的强大类型推导能力，在很多时候我们并不需要显式地去声明类型，但是显然函数并不在此列，必须手动为函数的所有参数和返回值指定类型，原因在于函数往往会作为 API 提供给你的用户，因此你的用户必须在使用时知道传入参数的类型和返回值类型。

与函数相反，闭包并不会作为 API 对外提供，因此它可以享受编译器的类型推导能力，无需标注参数和返回值的类型。

为了增加代码可读性，有时候我们会显式地给类型进行标注，出于同样的目的，也可以给闭包标注类型：

```
let sum = |x: i32, y: i32| -> i32 {
    x + y
}
```

与之相比，不标注类型的闭包声明会更简洁些：`let sum = |x, y| x + y`，需要注意的是，针对 `sum` 闭包，如果你只进行了声明，但是没有使用，编译器会提示你为 `x, y` 添加类型标注，因为它缺乏必要的上下文：

```
let sum  = |x, y| x + y;
let v = sum(1, 2);
```

这里我们使用了 `sum`，同时把 `1` 传给了 `x`， `2` 传给了 `y`，因此编译器才可以推导出 `x,y` 的类型为 `i32`。

下面展示了同一个功能的函数和闭包实现形式：

```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

可以看出第一行的函数和后面的闭包其实在形式上是非常接近的，同时三种不同的闭包也展示了三种不同的使用方式：省略参数、返回值类型和花括号对。

虽然类型推导很好用，但是它不是泛型，**当编译器推导出一种类型后，它就会一直使用该类型**：

```
let example_closure = |x| x;

let s = example_closure(String::from("hello"));
let n = example_closure(5);
```

首先，在 `s` 中，编译器为 `x` 推导出类型 `String`，但是紧接着 `n` 试图用 `5` 这个整型去调用闭包，跟编译器之前推导的 `String` 类型不符，因此报错：

```
error[E0308]: mismatched types
 --> src/main.rs:5:29
  |
5 |     let n = example_closure(5);
  |                             ^
  |                             |
  |                             expected struct `String`, found integer // 期待String类
型，却发现一个整数
  |                             help: try using a conversion method: `5.to_string()`
```

结构体中的闭包

假设我们要实现一个简易缓存，功能是获取一个值，然后将其缓存起来，那么可以这样设计：

- 一个闭包用于获取值
- 一个变量，用于存储该值

可以使用结构体来代表缓存对象，最终设计如下：

```rust
struct Cacher<T>
where
    T: Fn(u32) -> u32,
{
    query: T,
    value: Option<u32>,
}
```

等等，我都跟着这本教程学完 Rust 基础了，为何还有我不认识的东东？ `Fn(u32) -> u32` 是什么鬼？别急，先回答你第一个问题：骚年，too young too naive，你以为 Rust 的语法特性就基础入门那一些吗？太年轻了！如果是长征，你才刚到赤水河。

其实，可以看得出这一长串是 `T` 的特征约束，再结合之前的已知信息： `query` 是一个闭包，大概可以推测出， `Fn(u32) -> u32` 是一个特征，用来表示 `T` 是一个闭包类型？Bingo，恭喜你，答对了！

那为什么不用具体的类型来标注 `query` 呢？原因很简单，每一个闭包实例都有独属于自己的类型，即使于两个签名一模一样的闭包，它们的类型也是不同的，因此你无法用一个统一的类型来标注 `query` 闭包。

而标准库提供的 `Fn` 系列特征，再结合特征约束，就能很好的解决了这个问题. `T: Fn(u32) -> u32` 意味着 `query` 的类型是 `T`，该类型必须实现了相应的闭包特征 `Fn(u32) -> u32`。从特征的角度来看它长得非常反直觉，但是如果从闭包的角度来看又极其符合直觉，不得不佩服 Rust 团队的鬼才设计。。。

特征 `Fn(u32) -> u32` 从表面来看，就对闭包形式进行了显而易见的限制：**该闭包拥有一个 `u32` 类型的参数，同时返回一个 `u32` 类型的值**。

> 需要注意的是，其实 Fn 特征不仅仅适用于闭包，还适用于函数，因此上面的 `query` 字段除了使用闭包作为值外，还能使用一个具名的函数来作为它的值

接着，为缓存实现方法：

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

    // 先查询缓存值 `self.value`，若不存在，则调用 `query` 加载
    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.query)(arg);
                self.value = Some(v);
                v
            }
        }
    }
}
```

上面的缓存有一个很大的问题：只支持 `u32` 类型的值，若我们想要缓存 `&str` 类型，显然就行不通了，因此需要将 `u32` 替换成泛型 `E`，该练习就留给读者自己完成，具体代码可以参考这里

捕获作用域中的值

在之前代码中，我们一直在用闭包的匿名函数特性（赋值给变量），然而闭包还拥有一项函数所不具备的特性：捕获作用域中的值。

```rust
fn main() {
    let x = 4;

    let equal_to_x = |z| z == x;

    let y = 4;

    assert!(equal_to_x(y));
}
```

上面代码中， `x` 并不是闭包 `equal_to_x` 的参数，但是它依然可以去使用 `x`，因为 `equal_to_x` 在 `x` 的作用域范围内。

对于函数来说，就算你把函数定义在 `main` 函数体中，它也不能访问 `x`：

```rust
fn main() {
    let x = 4;

    fn equal_to_x(z: i32) -> bool {
        z == x
    }

    let y = 4;

    assert!(equal_to_x(y));
}
```

报错如下：

```
error[E0434]: can't capture dynamic environment in a fn item // 在函数中无法捕获动态的环境
 --> src/main.rs:5:14
  |
5 |         z == x
  |              ^
  |
  = help: use the `|| { ... }` closure form instead // 使用闭包替代
```

如上所示，编译器准确地告诉了我们错误，甚至同时给出了提示：使用闭包来替代函数，这种聪明令我有些无所适从，总感觉会显得我很笨。

闭包对内存的影响

当闭包从环境中捕获一个值时，会分配内存去存储这些值。对于有些场景来说，这种额外的内存分配会成为一种负担。与之相比，函数就不会去捕获这些环境值，因此定义和使用函数不会拥有这种内存负担。

三种 Fn 特征

闭包捕获变量有三种途径，恰好对应函数参数的三种传入方式：转移所有权、可变借用、不可变借用，因此相应的 `Fn` 特征也有三种：

1. `FnOnce` ，该类型的闭包会拿走被捕获变量的所有权。 `Once` 顾名思义，说明该闭包只能运行一次：

```
fn fn_once<F>(func: F)
where
    F: FnOnce(usize) -> bool,
{
    println!("{}", func(3));
    println!("{}", func(4));
}

fn main() {
    let x = vec![1, 2, 3];
    fn_once(|z|{z == x.len()})
}
```

仅实现 `FnOnce` 特征的闭包在调用时会转移所有权，所以显然不能对已失去所有权的闭包变量进行二次调用：

```
error[E0382]: use of moved value: `func`
 --> src\main.rs:6:20
  |
1 | fn fn_once<F>(func: F)
  |                ---- move occurs because `func` has type `F`, which does not
implement the `Copy` trait
                 // 因为`func`的类型是没有实现`Copy`特性的 `F`，所以发生了所有权的转移
...
5 |     println!("{}", func(3));
  |                    ------- `func` moved due to this call // 转移在这
6 |     println!("{}", func(4));
  |                    ^^^^ value used here after move // 转移后再次用
  |
```

这里面有一个很重要的提示，因为 `F` 没有实现 `Copy` 特征，所以会报错，那么我们添加一个约束，试试实现了 `Copy` 的闭包：

```
fn fn_once<F>(func: F)
where
    F: FnOnce(usize) -> bool + Copy,// 改动在这里
{
    println!("{}", func(3));
    println!("{}", func(4));
}

fn main() {
    let x = vec![1, 2, 3];
    fn_once(|z|{z == x.len()})
}
```

上面代码中， `func` 的类型 `F` 实现了 `Copy` 特征，调用时使用的将是它的拷贝，所以并没有发生所有权的转移。

```
true
false
```

如果你想强制闭包取得捕获变量的所有权，可以在参数列表前添加 `move` 关键字，这种用法通常用于闭包的生命周期大于捕获变量的生命周期时，例如将闭包返回或移入其他线程。

```rust
use std::thread;
let v = vec![1, 2, 3];
let handle = thread::spawn(move || {
    println!("Here's a vector: {:?}", v);
});
handle.join().unwrap();
```

2. `FnMut`，它以可变借用的方式捕获了环境中的值，因此可以修改该值：

```rust
fn main() {
    let mut s = String::new();

    let update_string =  |str| s.push_str(str);
    update_string("hello");

    println!("{:?}",s);
}
```

在闭包中，我们调用 `s.push_str` 去改变外部 `s` 的字符串值，因此这里捕获了它的可变借用，运行下试试：

```
error[E0596]: cannot borrow `update_string` as mutable, as it is not declared as
mutable
 --> src/main.rs:5:5
  |
4 |     let update_string =  |str| s.push_str(str);
  |         -------------          - calling `update_string` requires mutable binding
due to mutable borrow of `s`
  |         |
  |         help: consider changing this to be mutable: `mut update_string`
5 |     update_string("hello");
  |     ^^^^^^^^^^^^^ cannot borrow as mutable
```

虽然报错了，但是编译器给出了非常清晰的提示，想要在闭包内部捕获可变借用，需要把该闭包声明为可变类型，也就是 `update_string` 要修改为 `mut update_string`：

```rust
fn main() {
    let mut s = String::new();

    let mut update_string =  |str| s.push_str(str);
    update_string("hello");

    println!("{:?}",s);
}
```

这种写法有点反直觉，相比起来前面的 `move` 更符合使用和阅读习惯。但是如果你忽略 `update_string` 的类型，仅仅把它当成一个普通变量，那么这种声明就比较合理了。

再来看一个复杂点的：

```rust
fn main() {
    let mut s = String::new();

    let update_string =  |str| s.push_str(str);

    exec(update_string);

    println!("{:?}",s);
}

fn exec<'a, F: FnMut(&'a str)>(mut f: F)  {
    f("hello")
}
```

这段代码中 `update_string` 没有使用mut关键字修饰，而上文提到想要在闭包内部捕获可变借用，需要用关键词把该闭包声明为可变类型。我们确实这么做了——— `exec(mut f: F)` 表明我们的 `exec` 接收的是一个可变类型的闭包。这段代码中 `update_string` 看似被声明为不可变闭包，但是 `exec(mut f: F)` 函数接收的又是可变参数，为什么可以正常执行呢?

rust不可能接受类型不匹配的形参和实参通过编译，我们提供的实参又是可变的，这说明 `update_string` 一定是一个可变类型的闭包，我们不妨看看rust-analyzer自动给出的类型标注：

```
let mut s: String = String::new();

let update_string: impl FnMut(&str) =  |str| s.push_str(str);
```

rust-analyzer给出的类型标注非常清晰的说明了 `update_string` 实现了 `FnMut` 特征。

为什么 `update_string` 没有用 `mut` 修饰却是一个可变类型的闭包？事实上， `FnMut` 只是trait的名字，声明变量为 `FnMut` 和要不要mut没啥关系， `FnMut` 是推导出的特征类型， `mut` 是rust语言层面的一个修饰符，用于声明一个绑定是可变的。Rust从特征类型系统和语言修饰符两方面保障了我们的程序正确运行。

我们在使用 `FnMut` 类型闭包时需要捕获外界的可变借用，因此我们常常搭配 `mut` 修饰符使用。但我们要始终记住，二者是相互独立的。

因此，让我们再回头分析一下这段代码：在 `main` 函数中，首先创建了一个可变的字符串 `s` ，然后定义了一个可变类型闭包 `update_string` ，该闭包接受一个字符串参数并将其追加到 `s` 中。接下来调用了 `exec` 函数，并将 `update_string` 闭包的所有权移交给它。最后打印出了字符串 `s` 的内容。

细心的读者可能注意到，我们在上文的分析中提到 `update_string` 闭包的所有权被移交给了 `exec` 函数。这说明 `update_string` 没有实现 `Copy` 特征，但并不是所有闭包都没有实现 `Copy` 特征，闭包自动实现 `Copy` 特征的规则是，只要闭包捕获的类型都实现了 `Copy` 特征的话，这个闭包就会默认实现 `Copy` 特征。

我们来看一个例子：

```
let s = String::new();
let update_string =  || println!("{}",s);
```

这里取得的是 `s` 的不可变引用，所以是能 `Copy` 的。而如果拿到的是 `s` 的所有权或可变引用，都是不能 `Copy` 的。我们刚刚的代码就属于第二类，取得的是 `s` 的可变引用，没有实现 `Copy` 。

```
// 拿所有权
let s = String::new();
let update_string = move || println!("{}", s);

exec(update_string);
// exec2(update_string); // 不能再用了

// 可变引用
let mut s = String::new();
let mut update_string = || s.push_str("hello");
exec(update_string);
// exec1(update_string); // 不能再用了
```

3. `Fn` 特征，它以不可变借用的方式捕获环境中的值 让我们把上面的代码中 `exec` 的 `F` 泛型参数类型修改为 `Fn(&'a str)`：

```
fn main() {
    let mut s = String::new();

    let update_string =  |str| s.push_str(str);

    exec(update_string);

    println!("{:?}",s);
}

fn exec<'a, F: Fn(&'a str)>(mut f: F)  {
    f("hello")
}
```

然后运行看看结果：

```
error[E0525]: expected a closure that implements the `Fn` trait, but this closure only
implements `FnMut`
 --> src/main.rs:4:26  // 期望闭包实现的是`Fn`特征，但是它只实现了`FnMut`特征
  |
4 |     let update_string =  |str| s.push_str(str);
  |                          ^^^^^^-^^^^^^^^^^^^^^
  |                          |     |
  |                          |     closure is `FnMut` because it mutates the variable
`s` here
  |                          this closure implements `FnMut`, not `Fn` //闭包实现的是
FnMut，而不是Fn
5 |
6 |     exec(update_string);
  |     ---- the requirement to implement `Fn` derives from here
```

从报错中很清晰的看出，我们的闭包实现的是 `FnMut` 特征，需要的是可变借用，但是在 `exec` 中却给它标注了 `Fn` 特征，因此产生了不匹配，再来看看正确的不可变借用方式：

```
fn main() {
    let s = "hello, ".to_string();

    let update_string =  |str| println!("{},{}",s,str);

    exec(update_string);

    println!("{:?}",s);
}

fn exec<'a, F: Fn(String) -> ()>(f: F)  {
    f("world".to_string())
}
```

在这里，因为无需改变 `s`，因此闭包中只对 `s` 进行了不可变借用，那么在 `exec` 中，将其标记为 `Fn` 特征就完全正确。

move 和 Fn

在上面，我们讲到了 `move` 关键字对于 `FnOnce` 特征的重要性，但是实际上使用了 `move` 的闭包依然可以使用 `Fn` 或 `FnMut` 特征。

因为，**一个闭包实现了哪种 Fn 特征取决于该闭包如何使用被捕获的变量，而不是取决于闭包如何捕获它们。** `move` 本身强调的就是后者，闭包如何捕获变量：

```
fn main() {
    let s = String::new();

    let update_string =  move || println!("{}",s);

    exec(update_string);
}

fn exec<F: FnOnce()>(f: F)  {
    f()
}
```

我们在上面的闭包中使用了 `move` 关键字，所以我们的闭包捕获了它，但是由于闭包获取了 `s` 的所有权，因此该闭包实现了 `FnOnce` 的特征。

但是假如我们将代码修改成下面这样，依然可以编译：

```
fn main() {
    let s = String::new();

    let update_string =  move || println!("{}",s);

    exec(update_string);
}

fn exec<F: Fn()>(f: F)  {
    f()
}
```

奇怪， 明明是闭包实现的是 `FnOnce` 的特征， 为什么编译器居然允许 `Fn` 特征通过编译呢?

三种 Fn 的关系

实际上，一个闭包并不仅仅实现某一种 `Fn` 特征，规则如下：

- 所有的闭包都自动实现了 `FnOnce` 特征，因此任何一个闭包都至少可以被调用一次
- 没有移出所捕获变量的所有权的闭包自动实现了 `FnMut` 特征
- 不需要对捕获变量进行改变的闭包自动实现了 `Fn` 特征

用一段代码来简单诠释上述规则：

```
fn main() {
    let s = String::new();

    let update_string =  || println!("{}",s);

    exec(update_string);
    exec1(update_string);
    exec2(update_string);
}

fn exec<F: FnOnce()>(f: F)  {
    f()
}

fn exec1<F: FnMut()>(mut f: F)  {
    f()
}

fn exec2<F: Fn()>(f: F)  {
    f()
}
```

虽然，闭包只是对 `s` 进行了不可变借用，实际上，它可以适用于任何一种 `Fn` 特征：三个 `exec` 函数说明了一切。强烈建议读者亲自动手试试各种情况下使用的 `Fn` 特征，更有助于加深这方面的理解。

关于第二条规则，有如下示例：

```rust
fn main() {
    let mut s = String::new();

    let update_string = |str| -> String {s.push_str(str); s };

    exec(update_string);
}

fn exec<'a, F: FnMut(&'a str) -> String>(mut f: F) {
    f("hello");
}
```

```
5 |     let update_string = |str| -> String {s.push_str(str); s };
  |                         ^^^^^^^^^^^^^^^^                  - closure is `FnOnce`
because it moves the variable `s` out of its environment
  |                                                           // 闭包实现了`FnOnce`，因为
它从捕获环境中移出了变量`s`
  |                         |
  |                         this closure implements `FnOnce`, not `FnMut`
```

此例中，闭包从捕获环境中移出了变量 `s` 的所有权，因此这个闭包仅自动实现了 `FnOnce`，未实现 `FnMut` 和 `Fn`。再次印证之前讲的**一个闭包实现了哪种 Fn 特征取决于该闭包如何使用被捕获的变量，而不是取决于闭包如何捕获它们**，跟是否使用 `move` 没有必然联系。

如果还是有疑惑？没关系，我们来看看这三个特征的简化版源码：

```rust
pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}

pub trait FnMut<Args> : FnOnce<Args> {
    extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait FnOnce<Args> {
    type Output;

    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
```

看到没？从特征约束能看出来 `Fn` 的前提是实现 `FnMut`， `FnMut` 的前提是实现 `FnOnce`，因此要实现 `Fn` 就要同时实现 `FnMut` 和 `FnOnce`，这段源码从侧面印证了之前规则的正确性。

从源码中还能看出一点： `Fn` 获取 `&self`， `FnMut` 获取 `&mut self`，而 `FnOnce` 获取 `self`。 在实际项目中，**建议先使用 `Fn` 特征**，然后编译器会告诉你正误以及该如何选择。

闭包作为函数返回值

看到这里，相信大家对于如何使用闭包作为函数参数，已经很熟悉了，但是如果要使用闭包作为函数返回值，该如何做?

先来看一段代码:

```
fn factory() -> Fn(i32) -> i32 {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

上面这段代码看起来还是蛮正常的，用 `Fn(i32) -> i32` 特征来代表 `|x| x + num`，非常合理嘛，肯定可以编译通过, 可惜理想总是难以照进现实，编译器给我们报了一大堆错误，先挑几个重点来看看:

```
fn factory<T>() -> Fn(i32) -> i32 {
  |                     ^^^^^^^^^^^^^^^ doesn't have a size known at compile-time // 该类型在编译器没有固定的大小
```

Rust 要求函数的参数和返回类型，必须有固定的内存大小，例如 `i32` 就是 4 个字节，引用类型是 8 个字节，总之，绝大部分类型都有固定的大小，但是不包括特征，因为特征类似接口，对于编译器来说，无法知道它后面藏的真实类型是什么，因为也无法得知具体的大小。

同样，我们也无法知道闭包的具体类型，该怎么办呢？再看看报错提示：

```
help: use `impl Fn(i32) -> i32` as the return type, as all return paths are of type
`[closure@src/main.rs:11:5: 11:21]`, which implements `Fn(i32) -> i32`
  |
8 | fn factory<T>() -> impl Fn(i32) -> i32 {
```

嗯，编译器提示我们加一个 `impl` 关键字，哦，这样一说，读者可能就想起来了， `impl Trait` 可以用来返回一个实现了指定特征的类型，那么这里 `impl Fn(i32) -> i32` 的返回值形式，说明我们要返回一个闭包类型，它实现了 `Fn(i32) -> i32` 特征。

完美解决，但是，在特征那一章，我们提到过， `impl Trait` 的返回方式有一个非常大的局限，就是你只能返回同样的类型，例如：

```
fn factory(x:i32) -> impl Fn(i32) -> i32 {

    let num = 5;

    if x > 1{
        move |x| x + num
    } else {
        move |x| x - num
    }
}
```

运行后，编译器报错：

```
error[E0308]: `if` and `else` have incompatible types
  --> src/main.rs:15:9
   |
12 | /     if x > 1{
13 | |         move |x| x + num
   | |         ---------------- expected because of this
14 | |     } else {
15 | |         move |x| x - num
   | |         ^^^^^^^^^^^^^^^^ expected closure, found a different closure
16 | |     }
   | |_____- `if` and `else` have incompatible types
   |
```

嗯，提示很清晰： `if` 和 `else` 分支中返回了不同的闭包类型，这就很奇怪了，明明这两个闭包长的一样的，好在细心的读者应该回想起来，本章节前面咱们有提到：就算签名一样的闭包，类型也是不同的，因此在这种情况下，就无法再使用 `impl Trait` 的方式去返回闭包。

怎么办？再看看编译器提示，里面有这样一行小字：

```
 = help: consider boxing your closure and/or using it as a trait object
```

哦，相信你已经恍然大悟，可以用特征对象！只需要用 `Box` 的方式即可实现：

```
fn factory(x:i32) -> Box<dyn Fn(i32) -> i32> {
    let num = 5;

    if x > 1{
        Box::new(move |x| x + num)
    } else {
        Box::new(move |x| x - num)
    }
}
```

至此，闭包作为函数返回值就已完美解决，若以后你再遇到报错时，一定要仔细阅读编译器的提示，很多时候，转角都能遇到爱。

闭包的生命周期

这块儿内容在进阶生命周期章节中有讲，这里就不再赘述，读者可移步此处进行回顾。

课后习题

Rust By Practice，支持代码在线编辑和运行，并提供详细的习题解答。