

13 闭包

闭包（Closure），走了这么远的路，终于见到一个熟悉的了。基本上和 JavaScript 中的箭头函数（Arrow Function）是一样的，它可以被存储到一个变量中，也可以从当前定义上下文捕获变量，也可以在定义中携带参数。

但是，从 Rust 本身而言，闭包是一种很特殊的数据结构，它近似于一个匿名的、包含了它所捕获的变量的信息的结构体。闭包的基本形式很简单：

```
1 | args | { }
```

和 JavaScript 的箭头函数类似，如果闭包中的代码只有一行，也可以省略花括号。

让我们从一个例子开始。上一节中讲到 `Option<T>` 枚举的时候，我们列出了它的常用方法：`unwrap()`、`unwrap_or()`。还有一个常用方法：`unwrap_or_else()`，这个方法的定义如下：

```
1 pub fn unwrap_or_else<F>(self, f: F) -> T
2 where
3     F: FnOnce() -> T,
```

从定义可以看出，`F` 是一个闭包函数，它没有任何入参，返回的类型是 `Option<T>` 中对应的 `T`。这个函数的含义是：如果 `Option<T>` 的值是 `Some<T>`，则返回 `T`，否则，调用传入的闭包函数 `f`，使用它的返回值作为 `T` 类型的值。

和 JavaScript 箭头函数的能力类似，Rust 中的闭包也可以捕获上下文中的变量。但是在 Rust 的闭包中，会根据实际的代码来决定是以值（移动）的方式捕获变量、以只读引用的方式捕获变量还是以可写引用的方式捕获变量。

下面这段代码，对于变量 `x`，闭包会捕获它的只读引用：

```

1 let x = 10;
2 let closure = || println!("{}", x);
3 closure();
4 // x 可以在外部作用域继续使用

```

如果我们在闭包中修改了外层上下文中变量的值，那么闭包会捕获一个可写引用。注意，这里的闭包本身也需要配合 `mut` 关键字：

```

1 let mut x = 10;
2 let mut closure = || {
3     x += 1;
4     println!("{}", x);
5 };
6 closure();
7 // x 现在是 11

```

例如，下面这段代码，由于使用了 `move` 关键字，变量 `s` 的值将会被移动到闭包中，也就是说会发生所有权转移：

```

1 fn main() {
2     let s = String::from("Hello world!");
3     let closure = move || {
4         let tp = (s, 1);
5         println!("{}: {}", tp.0, tp.1);
6     };
7     closure();
8     // s 的所有权已经被移动到闭包中，后面的代码无法再使用 s
9     // println!("{}: {}", s, 1);
10 }

```

所以，在 Rust 中，闭包分为以下三种类型（trait），如果你的 IDE 有镶嵌提示的功能，那么你会看到提示：

- `Fn` 闭包，不捕获上下文变量，或者捕获上下文变量的只读引用
- `FnMut` 闭包，捕获上下文变量的可写引用

- `FnOnce` 闭包，上下文变量拥有的值的所有权将会被转移到闭包中，这也是为什么它只能被调用一次（Once），因为只有一次机会转移所有权，再次调用的时候就会出错了

闭包通常和集合+迭代器（Iterator）一起工作的，即使是 `Option<T>` 和 `Result<T, E>` 也是实现了 `IntoIterator` trait 的。