

现代编程思想

命令式编程

Hongbo Zhang

函数式编程

- 到此为止，我们介绍的可以归类于函数式编程的范畴
 - 对每一个输入，有着固定的输出
 - 对于标识符，我们可以直接用它所对应的值进行替代——引用透明性
- 开发实用的程序，我们需要一些计算之外的“副作用”
 - 进行输入输出
 - 修改内存中的数据等
 - 这些副作用可能导致多次执行的结果不一致

引用透明性

- 我们可以定义如下数据绑定和函数

```
1. let x: Int = 1 + 1
2. fn square(x: Int) -> Int { x * x }
3. let z: Int = square(x) // 4
```

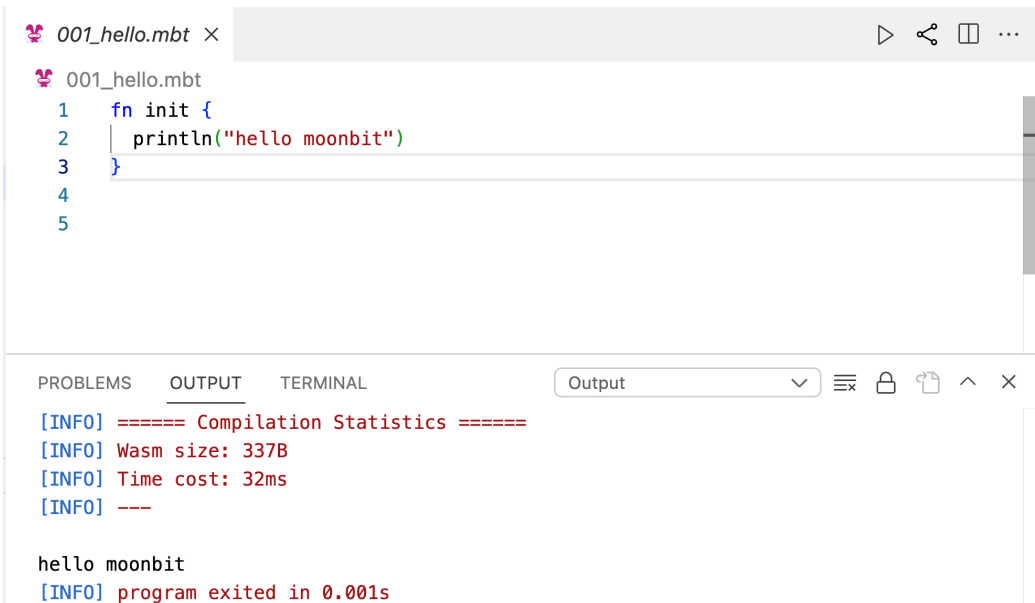
- 我们可以将 `square` 与 `x` 直接用对应的值替换而不改变结果

```
1. let z: Int = { 2 * 2 } // 4
```

- 引用透明性可以易于理解

- 函数 `print` 允许我们输出一个字符串，例如 `print("hello moonbit")`
- 月兔中可以通过 `init` 代码块来定义初始化指令

```
1. fn init {  
2.     println("hello moonbit") // 函数名中的ln代表换行  
3. }
```



The screenshot shows the MoonBit IDE interface. The top part is a code editor with a file named `001_hello.mbt`. The code inside is:

```
1 fn init {  
2     println("hello moonbit")  
3 }  
4  
5
```

The bottom part of the IDE is a terminal window with tabs for `PROBLEMS`, `OUTPUT`, and `TERMINAL`. The `OUTPUT` tab is selected, showing the following text:

```
[INFO] ===== Compilation Statistics =====  
[INFO] Wasm size: 337B  
[INFO] Time cost: 32ms  
[INFO] ---  
  
hello moonbit  
[INFO] program exited in 0.001s
```

命令与副作用

- 输出命令可能会破坏引用透明性

```
1. fn square(x: Int) -> Int { x * x }
2. fn init {
3.     let x: Int = {
4.         println("hello moonbit") // <-- 我们首先执行命令，进行输出
5.         1 + 1 // <-- 之后，我们以表达式块最后的值作为表达式块的值
6.     }
7.     let z: Int = square(x) // 4, 输出一次
8. }
```

PROBLEMS

1

OUTPUT

TERMINAL

Output



```
[INFO] ===== Compilation Statistics =====
[INFO] Wasm size: 359B
[INFO] Time cost: 15ms
[INFO] ---
```

```
hello moonbit
```

```
[INFO] program exited in 0.003s
```

命令与副作用

- 我们不一定可以放心替换，因此会增大程序理解难度

```
1. fn init {  
2.     let z: Int = {  
3.         println("hello moonbit"); // <-- 进行了第一次输出  
4.         1 + 1 // <-- 获得值: 2  
5.     } * {  
6.         println("hello moonbit"); // <-- 进行了第二次输出  
7.         1 + 1 // <-- 获得值: 2  
8.     } // 4, 输出两次  
9. }
```

PROBLEMS 1 OUTPUT TERMINAL

Output



```
[INFO] ===== Compilation Statistics =====  
[INFO] Wasm size: 350B  
[INFO] Time cost: 7ms  
[INFO] ---
```

```
hello moonbit  
hello moonbit
```

```
[INFO] program exited in 0.001s
```

单值类型

- 我们之前已经介绍过单值类型 `Unit`
 - 它仅有一个值: `()`
- 以 `Unit` 为运算结果类型的函数或命令一般有副作用
 - `fn print(String) -> Unit`
 - `fn println(String) -> Unit`
- 命令的类型也是单值类型

```
1. fn do_nothing() -> Unit { // 返回值为单值类型时可以省略返回类型声明
2.   let _x = 0 // 结果为单值类型, 符合函数定义
3. }
```

变量

- 在月兔中，我们可以在代码块中用 `let mut` 定义临时变量

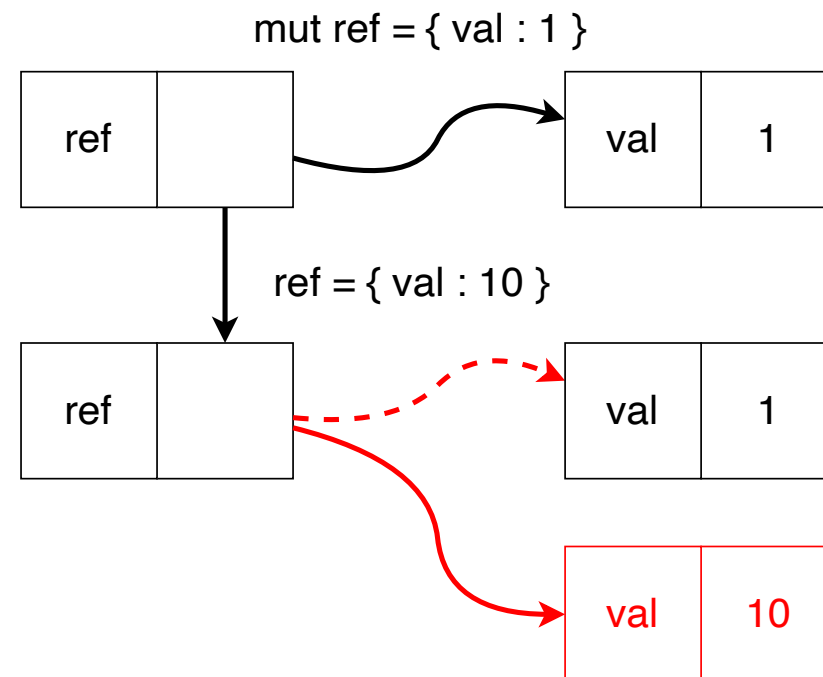
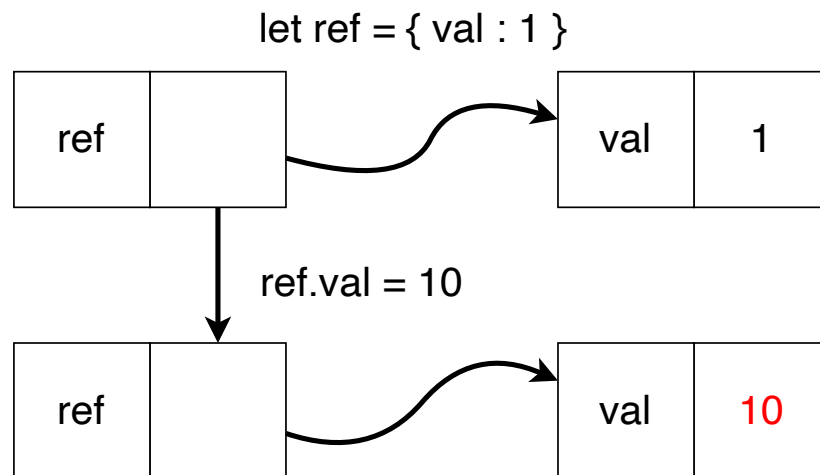
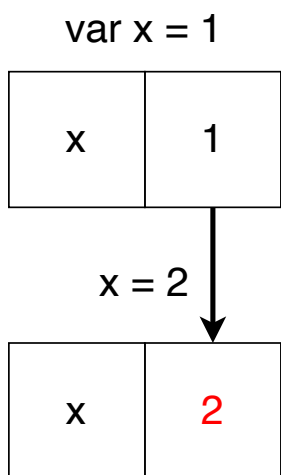
```
1. test {  
2.     let mut x = 1  
3.     x = 10 // 赋值操作是一个命令  
4. }
```

- 在月兔中，结构体的字段默认不可变，我们也允许可变的字段，需要用 `mut` 标识

```
1. struct Ref[T] { mut val : T }  
2.  
3. fn init {  
4.     let ref: Ref[Int] = { val: 1 } // ref 本身只是一个数据绑定  
5.     ref.val = 10 // 我们可以修改结构体的字段  
6.     println(ref.val.to_string()) // 输出 10  
7. }
```


变量

- 我们可以将带有可变字段的结构体看作是引用



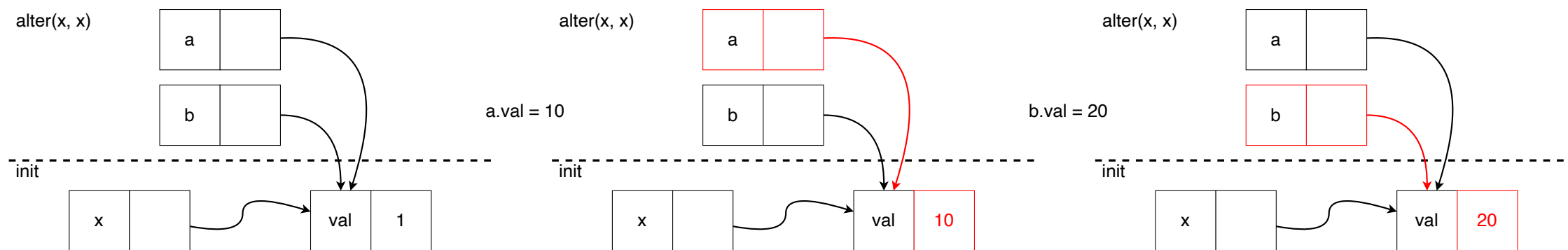
别名

- 指向相同的可变数据结构的两个标识符可以看作是别名

```
1. fn alter(a: Ref[Int], b: Ref[Int]) -> Unit {  
2.     a.val = 10  
3.     b.val = 20  
4. }  
5.  
6. fn init {  
7.     let x: Ref[Int] = { val : 1 }  
8.     alter(x, x)  
9.     println(x.val.to_string()) // x.val的值将会被改变两次  
10. }
```

别名

- 指向相同的可变数据结构的两个标识符可以看作是别名



- 可变变量需要小心处理

循环

- 利用变量，我们可以定义循环

```
1. <定义变量及初始值>
2. while <针对变量判断是否继续循环> {
3.     <需要重复执行的命令>
4.     <对变量进行迭代>
5. }
```

- 例如，我们可以反复执行n次输出操作

```
1. let mut i = 0
2. while i < 2 {
3.     println("Output")
4.     i = i + 1
5. } // 重复输出2次
```

循环

- 我们进入循环时
 - 判断是否满足继续循环的条件
 - 执行命令
 - 对变量进行迭代
 - 重复以上过程
- 例如

```
1. let mut i = 0 // <-- 此时 i 等于 0
2. while i < 2 { // <-- 此处, 我们判断 i < 2 是否为真
3.     println("Output") // <-- 0 < 2, 因此继续执行, 输出第一次
4.     i = i + 1 // <-- 此时, 我们执行 i = i + 1
5. }
```

循环

- 我们进入循环时
 - 判断是否满足继续循环的条件
 - 执行命令
 - 对变量进行迭代
 - 重复以上过程
- 例如

```
1. // 此时 i 等于 1
2. while i < 2 { // <-- 此处, 我们判断 i < 2 是否为真
3.     println("Output") // <-- 1 < 2, 因此继续执行, 输出第二次
4.     i = i + 1 // <-- 此时, 我们执行 i = i + 1
5. }
```

循环

- 我们进入循环时
 - 判断是否满足继续循环的条件
 - 执行命令
 - 对变量进行迭代
 - 重复以上过程
- 例如

```
1. // 此时 i 等于 2
2. while i < 2 { // <-- 此处，我们判断 i < 2 是否为真，结果为假
3. } // <-- 跳过
4. // <-- 继续后续执行
```

调试器

- 月兔的调试器允许我们在运行中看到实时的运行数据，更好理解运行过程

The screenshot displays the MoonBit debugger interface. The left sidebar shows the project structure with files like `main.mbt` and `main.wasm`. The main editor shows the source code of a Fibonacci function. The right sidebar provides a detailed view of the current execution state, including the stack, local variables, and the call stack.

Source Code (main.mbt):

```

1  fn fib(n : Int) -> Int {
2      var acc1 = 0
3      var acc2 = 1
4
5      debugger()
6      var i = 0
7      while i < n, i = i + 1 {
8          let t = acc1 + acc2
9          acc1 = acc2
10         acc2 = t
11     }
12     return acc1
13 }
14
15 fn debugger() = "js" "debugger"
16 fn log(n: Int) = "js" "log"
17
18 fn init {
19     let n = fib(39)
20     log(n)
21 }
22

```

Debugger State (Right Sidebar):

- 调试程序已暂停** (Debugging program paused)
- 监视** (Watch)
- 断点** (Breakpoints)
 - ☐ 遇到未捕获的异常时暂停 (Pause on uncaught exception)
 - ☐ 在遇到异常时暂停 (Pause on exception)
- 作用域** (Scope)
- 表达式** (Expression)
 - stack:** Stack {}
- 本地** (Local)
 - \$acc1/1:** i32 {value: 0}
 - \$acc2/2:** i32 {value: 1}
 - \$i/3:** i32 {value: 0}
 - \$n/4:** i32 {value: 39}
 - \$t/5:** i32 {value: 0}
- 模块** (Module)
- 调用堆栈** (Call Stack)

\$\$\$moonbit-debugging-example/main.fib.fn/2	main.mbt:7
\$*init*/4	main.mbt:19
(匿名)	(索引) :26

Bottom Status Bar:

{ } 第 7 行, 第 9 列 (来自 main.wasm) 覆盖率: 不适用

循环与递归

- 事实上，循环与递归是等价的

```
1. let mut <变量> = <初始值>
2. while <判断是否继续循环> {
3.     <需要重复执行的命令>
4.     <对变量进行迭代>
5. }
```

- 利用可变变量的情况下可以写成

```
1. fn loop_(<参数>) -> Unit {
2.     if <判断是否继续循环> {
3.         <需要重复执行的命令>
4.         loop_(<迭代后的参数>)
5.     } else { () }
6. }
7. loop_(<初始值>)
```

循环与递归

- 例如下述两段代码执行效果相同

```
1. let mut i = 0
2. while i < 2 {
3.     println("Hello!")
4.     i = i + 1
5. }
```

```
1. fn loop_(i: Int) -> Unit {
2.     if i < 2 {
3.         println("Hello!")
4.         loop_(i + 1)
5.     } else { () }
6. }
7. loop_(0)
```

循环流的控制

- 循环的时候，可以提前中止循环，或是跳过后续命令的执行
 - `break` 指令可以中止循环
 - `continue` 指令可以跳过后续运行，直接进入下一次循环

```
1. fn print_first_3() -> Unit {  
2.     let mut i = 0  
3.     while i < 10 {  
4.         if i == 3 {  
5.             break // 跳过从3开始的情况  
6.         } else {  
7.             println(i.to_string())  
8.         }  
9.         i = i + 1  
10.    }  
11. }
```

循环流的控制

- 循环的时候，可以提前中止循环，或是跳过后续命令的执行
 - `break` 指令可以中止循环
 - `continue` 指令可以跳过后续运行，直接进入下一次循环

```
1. fn print_skip_3() -> Unit {  
2.     let mut i = 0  
3.     while i < 10 {  
4.         if i == 3 {  
5.             continue // 跳过3  
6.         } else { () }  
7.         println(i.to_string())  
8.         i = i + 1  
9.     }  
10. }
```

月兔的检查

- 月兔会检查一个变量是否被修改，可以避免出现循环忘记加迭代条件

```
001_hello.mbt Error (warning 015): The mutability of 'i' is never used.
001_hello.mbt Int
1 fn init View Problem (\F8) Quick Fix... (⌘.)
2   var i = 0
3   while i < 5 {
4     println(i)
5   }
6 }
```

- 月兔也会检查函数返回结果是否与类型声明相同，可以避免错误的返回类型声明

```
001_hello.mbt 1 ×
001_hello.mbt > plus_one
1 fn plus_one(i: Int) {
2   i + 1
3 }
Expr Type Mismatch
  has type : Int
  wanted   : Unit
(Int, Int) -> Int
View No quick fixes
```

- 使用场景广泛
 - 直接操作程序外环境，如硬件等
 - 一些情况下性能更好，如随机访问数组等
 - 可以构建部分复杂数据结构，如图
 - 重复利用空间（原地修改）
- 可变数据并不总是与引用透明性冲突

```
1. fn fib_mut(n: Int) -> Int { // 对于相同输入，总是有相同输出
2.   let mut acc1 = 0; let mut acc2 = 1; let mut i = 0
3.   while i < n {
4.     let t = acc1 + acc2; acc1 = acc2; acc2 = t
5.     i = i + 1
6.   }
7.   acc1
8. }
```

总结

本章节初步接触了命令式编程，包括

- 如何使用命令
- 如何使用变量
- 如何使用循环等