第五章：函数

5.1 函数声明

在Yak语言中，函数本质上是一种“值”。笔者推荐您使用如下模式来声明函数：

```
add = func(a, b) {
    return a + b
}
println(add(1, 2))  // 输出: 3
```

在上述代码中，笔者通过函数字面量声明了一个函数，并将其赋值给变量 `add` 。与传统的函数声明方式相比，这里的函数字面量（即函数的定义）并没有指定函数名，而是当我们将此函数赋值给 `add` 后，您可以通过 `add` 来调用这个函数。在Yak中，所有用户定义的函数都是匿名的，也就是说，它们在定义时没有名称，直到被赋值给某个变量。

这个函数字面量表示了一个类型（签名）为func(a,b)的函数。从函数的类型上就可以看出来，这个函数需要给出两个入参才可以调用，而函数的返回值并不在函数类型上。

为了适应不同用户的使用习惯，Yak语言允许使用 `fn` 或 `def` 作为 `func` 关键字的替代，它们之间的用法是完全相同的，同时Yak也支持命名函数的声明方式，所以下面的代码案例中的函数定义都是可以直接使用的：

```
func abc() {
    println("Hello World Named-Function!")
}

// 使用 def 关键字来定义函数，和 func 作用完全相同
def (){ println("Function Defined with def keyword") }
def namedDefFunction() {
    println("Hello World Named-Function! With DEF!")
}

// 使用 fn 来定义函数，和 def, func 完全相同
fn (){ println("Function Defined with fn keyword") }
fn namedFn() {
    println("Hello World Named-Function! With FN!")
}
```

5.2 函数调用与返回

在Yak语言中，函数调用非常直观。您可以在函数定义后使用圆括号和参数列表来调用它，或者在定义函数的同时立即调用它，如下所示：

```
// 标准使用：使用一个变量来接收函数，并调用
hello = func(s){
    println(s)
}
hello("123")

// 函数本身是一个值，因此可以直接使用
func(s){
    println(s)
}("123")

// 立即执行的无参函数
func{
    println("123")
}
```

函数可以在被定义后使用圆括号传入参数列表进行调用，也可以在定义的时候直接进行调用。上面的例子中第三种写法比较特殊，它省略了空参数列表，适用于无参数的函数，使用这种写法定义的函数在定义后马上就会被调用。

处理返回值

Yak语言的函数支持灵活返回值机制，如下所示：

```
addsub = func(a, b) {
    return a + b, a - b
}
println(addsub(1, 2)) // [3, -1]
```

Yak语言中的多返回可以看作是返回了一个切片。当我们使用一个变量去接收多个返回值时，它表现为接收到了一个由这几个返回值组成的切片；当我们使用与返回值数量等量的变量去接收一个多返回值函数的返回值时，这个切片会被解构，并按照顺序赋值到每一个变量中。

- **单一变量接收**：如果您使用一个变量来接收多返回值函数的结果，那么这个变量将存储一个列表，列表中包含了所有的返回值。

```
result = addsub(1, 2)
println(result) // 输出: [3, -1]
```

- **多变量接收**：如果您使用多个变量来接收返回值，那么您必须使用与返回值数量相等的变量，否则程序将报错。

```
sum, diff = addsub(1, 2)
println(sum)  // 输出: 3
println(diff) // 输出: -1
```

如果您尝试使用不匹配数量的变量接收多个返回值，Yak语言将抛出一个错误，因为它无法将不确定数量的返回值分配给固定数量的变量。

```
sum = addsub(1, 2) // 正确: sum 是一个列表 [3, -1]
sum, diff, another = addsub(1, 2) // 错误: 返回值有两个，但尝试分配给三个变量

/*
OUTPUT:

Panic Stack:
File "/var/folders/....yak", in __yak_main__
--> 9 sum, diff, another = addsub(1, 2) // 错误: 返回值有两个，但尝试分配给三个变量

YakVM Panic: multi-assign failed: left value length[3] != right value length[2]
*/
```

- **无返回值：**如果一个函数没有返回值，我们使用一个变量去接收他的返回值时会得到一个 `nil`

函数不一定需要返回值。如果一个函数没有明确的返回语句，或者返回语句没有任何跟随的值，那么这个函数就被视为没有返回值。在这种情况下，如果您尝试使用一个变量去接收这个函数的返回值，那么您将得到一个特殊的值 `nil` ，表示“无值”或“空值”。

```
func noReturn() {
    println("This function does not return a value.")
}
result = noReturn()
println(result) // 输出: nil

/*
OUTPUT:

```

```
This function does not return a value.
<nil>
*/
```

在上面的例子中，`noReturn` 函数执行了一个打印操作，但没有返回任何值。当我们尝试将其“返回值”赋给变量 `result` 时，`result` 的值是 `nil`。

这种特性在处理函数返回值时非常有用，因为它允许您区分函数是有意返回空值，还是根本没有返回值。您可以使用 `nil` 来进行条件检查，以确定是否有值返回：

```
noReturn = func() {
    println("NoReturn Function Executed")
}

result = noReturn()
if result != nil {
    println("Function returned a value.")
} else {
    println("Function did not return a value.")
}

/*
OUTPUT:

NoReturn Function Executed
Function did not return a value.
*/
```

虽然Yak语言支持多返回值，也不限制返回值的数量，但是我们希望用户在编程时，确保对于一个函数的多个出口具有相同数量的返回值。在实际编程中，理解函数返回值的行为以及如何处理不同情况的返回值是至关重要的。这不仅有助于编写更健壮的代码，而且有助于调试和排除故障。

5.3 函数参数

函数作为一种“值”，我们自然也可以将函数像变量一样传递。当我使用一个函数作为另一个函数的参数的时候，我们就可以称这个参数为函数参数。通过这种方式我们可以在外部定义一种运算，并将运算传递到其他地方。

```
AfterFunc = func(dur,f){
    time.Sleep(dur)
    f()
}
```

```
5
6  println(now())
7  AfterFunc(2, func(){
8      println("2 seconds passed")
9  })
10 println(now())
```

在这个例子中，我们定义了一个在两秒后输出“2 seconds passed”的函数，并将其作为 `AfterFunc` 的参数传入。 `AfterFunc` 函数等待两秒后执行了我们传入的函数。上述代码执行的结果为：

```
1 2023-11-09 11:34:34.538482 +0800 CST m=+0.181281751
2 2 seconds passed
3 2023-11-09 11:34:36.543558 +0800 CST m=+2.186374751
```

作为参数传递，函数也可以作为返回值传递也可以组成数组等，但是读者需要记住的是，Yak中的函数本质上就是一个“值”，传递的时候和传递“字符串”或者“数字”没有任何不同。

5.4 函数的可变参数

在函数定义中，可变参数允许您传递任意数量的参数。在Yak语言中，您可以通过在参数名后加上省略号 `...` 来指定一个可变参数。这告诉Yak语言该函数可以接收任意数量的参数，并且这些参数将作为一个数组传递给函数。

5.4.1 定义可变参数函数

下面是一个使用可变参数的函数示例，它可以接收任意数量的数值参数，并计算它们的和：

```
1  sum = func(numbers...) {
2      total = 0
3      for number in numbers {
4          total += number
5      }
6      return total
7  }
8  // 使用可变参数函数
9  println(sum(1, 2, 3))  // 输出: 6
10 println(sum())         // 输出: 0
```

在上述代码中， `sum` 函数可以接收任意数量的参数。参数 `numbers` 在函数内部作为一个数组处理，可以通过循环来遍历所有的元素。

5.4.2 混合使用固定参数和可变参数

在 Yak 语言中，您还可以在函数中混合使用固定参数和可变参数。固定参数需要在可变参数之前声明。这是一个例子：

```
func sum(first, rest...) {
    total = first
    for number in rest {
        total += number
    }
    return total
}
// 使用混合参数函数
println(sum(12))           // 输出: 12
println(sum(12, 3, 5, 6))  // 输出: 26
```

在这个例子中，`sum` 函数有一个固定参数 `first` 和一个可变参数 `rest`。当调用 `sum` 函数时，至少需要一个参数（对应 `first`），其他的参数（如果有的话）将被收集到 `rest` 数组中。

5.4.3 注意事项

- 当您调用包含固定参数和可变参数的函数时，必须保证所有固定参数都被正确赋值。
- 可变参数必须是函数签名中的最后一个参数，因为它负责收集所有剩余的参数。
- 在函数体内，可变参数表现为一个数组，您可以使用循环或其他数组操作来处理它。

通过使用可变参数，您的函数将具有更大的灵活性，并能够处理更多的使用场景。这在创建通用函数或API时尤其有用，因为您可以允许用户根据需要传递任意数量的参数。

5.5 箭头函数

箭头函数是一种在许多现代编程语言中都存在的功能，它提供了一种简洁的方式来定义函数。

> 在 Yak 语言中，箭头函数的语法与 ECMAScript 类似，但有一个关键的区别：Yak 中的箭头函数不需要 `this` 上下文。

5.5.1 基本语法与定义

箭头函数通过使用 `=>` 符号来创建，左边是参数，右边是函数体：

```
arrowFunction = a => a + 1
```

这个箭头函数接收一个参数 `a` 并返回 `a + 1`。

箭头函数可以在参数中设置多个参数，当然也可以没有参数，读者可以参考下面案例：

```
arrowFunction2 = (a, b) => a + b
arrowFunction3 = () => 1 + 1
```

除此之外，箭头函数的 `=>` 后面不一定只能跟表达式，用户可以直接输入一个代码块儿，让他变成一个真实的“函数代码块”，只需要用 `{` 和 `}` 包围起来即可，使用 `return` 来作为语句返回值：

```
arrowFunction4 = (a, b) => {
    sum = a + b
    return sum
}
```

因此，读者也可直接使用这种定义形式来定义函数，非常简洁。箭头函数作为函数来说，仍然是支持函数的定义和使用的，因此普通函数定义可变参数的行为，对箭头函数仍然适用：

```
arrowFunctionWithRest = (args...) => {
    // 你可以在这里处理 args，它是一个包含所有传递给函数的参数的数组
    for arg in args {
        println(arg)
    }
}
arrowFunctionWithRest(1,2,3)

/*
OUTPUT:

1
2
3
*/
```

类似地，我们还可以使用混合固定和可变参数来定义箭头函数的参数：

```
arrowFunctionMixed = (fixed1, fixed2, rest...) => {
    println("Fixed parameters:", fixed1, fixed2)
    println("Rest parameters:", rest)
}

arrowFunctionMixed("a", "b", 1, 2, 3) // 将输出 "a", "b" 和 [1, 2, 3]
```

```
7
8  /*
9  OUTPUT:
10
11 Fixed parameters: a b
12 Rest parameters: [1 2 3]
13 */
```

在这个例子中，`arrowFunctionMixed` 接受两个固定参数 `fixed1` 和 `fixed2`，以及后面跟随的任意数量的参数 `rest`。

这样的功能使得箭头函数在处理不确定数量的参数时非常灵活，这在编写通用函数或者需要聚合多个参数的情况下非常有用。

5.5.2 调用箭头函数

箭头函数非常适合用作回调函数或任何需要简洁函数的场合：

```
1 result = ["a", 1, 2, 3].Filter(i => typeof(i) == int)
2 println(result) // 输出: [1 2 3]
```

在这个例子中，箭头函数被用作 `filter` 方法的参数，用于筛选数组中的整数类型元素。

笔者非常推荐当函数参数需要表达一个简单逻辑的时候将箭头函数作为函数参数使用。

5.6 函数的闭包特性

在 Yak 语言中，闭包是一种强大的功能，它允许函数捕获并包含其创建时的上下文环境。这意味着即使外部函数已经返回，闭包仍然能够访问和操作外部函数中的变量。闭包的这种能力让它们在编程中非常有用，尤其是在构建有状态的函数时。

让我们深入探讨闭包，并了解如何在 Yak 语言中使用它们。

5.6.1 闭包的定义与特性

闭包是指那些能够访问并操作其创建时词法作用域中的变量的函数。与普通函数相比，闭包函数是“有状态的”。这意味着闭包函数在多次调用之间可以保留状态信息，这些信息通常存储在它们的词法环境中。

5.6.2 创建闭包

在 Yak 语言中创建闭包的过程非常直接。你只需要在一个函数内部定义另一个函数，并返回它。内部定义的函数将捕获并使用其外部函数的变量。来看一个例子：

```
IntGeneratorFactory = func(a) {
    return func() {
        a = a + 1
        println(a)
    }
}
```

在这个例子中，`IntGeneratorFactory`是一个函数，它接受一个参数`a`，并返回一个新的函数。这个新函数能够访问并修改`a`的值，并在每次调用时打印出`a`的新值。

5.6.3 使用闭包

当我们使用`IntGeneratorFactory`函数时，每次调用都会创建一个新的闭包，每个闭包都有自己的状态：

```
IntGenerator1 = IntGeneratorFactory(0) // 使用参数0初始化状态
IntGenerator2 = IntGeneratorFactory(10) // 使用参数10初始化状态
```

这里，`IntGenerator1`和`IntGenerator2`是两个不同的闭包，它们分别捕获了不同的初始状态。

当我们调用这些闭包时，可以看到它们各自的状态是如何独立变化的：

```
IntGenerator1() // 输出: 1
IntGenerator1() // 输出: 2
IntGenerator1() // 输出: 3
IntGenerator2() // 输出: 11
IntGenerator2() // 输出: 12
IntGenerator2() // 输出: 13
```

每次调用`IntGenerator1`或`IntGenerator2`时，它们内部的`a`变量都会增加。这显示了闭包确实是有状态的，它们记住了之前的调用状态。

闭包在编程中的应用非常广泛，从数据隐藏和封装到函数工厂和模块化设计，闭包都发挥着重要作用。在Yak语言中，理解和掌握闭包可以帮助你编写更加高效和强大的代码。

闭包不仅仅是函数，它们是记忆了创建它们的环境的函数。这种能力使得闭包成为一个非常强大的构建，特别是在你需要生成具有私有状态的函数时。在 Yak 语言中，利用闭包可以优雅地构建复杂的功能，同时保持代码的清晰和模块化。

希望这个简介能帮助你理解闭包在 Yak 语言中的工作原理和它们的强大之处。现在，你已经有了使用闭包来构建有状态函数的知识基础，可以开始探索它们在实际编程中的应用了。