SwiftCon China 2016

www.swiftconchina.com

SwiftCon
www.swiftconchina.com
x
Think
ThinkInLAMP.com

从数学函数角度理解函数式编程

dingfeng

20160423

```
//数学函数

f(x) = 1 + x


f(1) = 1 + 1 = 2
f(3) = 1 + 3 = 4
```

```
//Swift 函数

func f(x: Int) -> Int {
 return x + 1
}


f(1)//2
f(3)//3
```

```
F(y) = 2y


F(1) = 2
F(3) = 6
```

```
func F(y: Int) -> Int {
    return 2 * y
}

F(1)//2
F(3)//6
```

F(y) = 2y

```
func F(y: Int) -> Int {
    return 2 * y
}
```

F(1.5) = 3.0

```
F(1.5)
//Cannot convert value of type 'Double'
//to expected argument type 'Int'
```

```
func F(y: Double) -> Double {
    return 2 * y
}

F(1)//2.0
F(3)//6.0
F(1.5)//3.0
```

```
f(x) = 1 + x
```

```
F(y) = 2y
```

```
   令 y = f(x)
   F(f(x)) = 2*f(x)

g(x) =  F(f(x)) =  2f(x)
= 2(x + 1)

令 x = 5
g(5) = F(f(5))=2 * 6 = 12
```

```
func f(x: Int) -> Int {
 return x + 1
}
```

```
func F(y: Int) -> Int {
    return 2 * y
}
```

```
func F(f: Int -> Int) -> Int -> Int {
    func ret(x: Int) -> Int {
       return 2 * f(x)
    }
    return ret
}

let g = F(f)

g(5)//12
```

高阶函数（**Higher-Order Functions**）

```
func F(f: Int -> Int) -> Int -> Int {
    func ret(x: Int) -> Int {
       return 2 * f(x)
    }
    return ret
}
```

闭包（Closure）

“a closure is a record storing a function together with an environment”
—wiki

“闭包是指函数有自由独立的变量，定义在闭包中的函数可以“记忆”它创建时候的环境。”
—mozilla

为什么强调闭包？我们知道引入高阶函数是有其缘由和作用的，那么闭包到底是怎么出现的？

【3】https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Closures

来具体的看一个闭包：

```
func F(f: Int -> Int) -> Int -> Int {
    func ret(x: Int) -> Int {
        return 2 * f(x)
    }
    return ret
}
```

```
func F(f: Int -> Int) -> Int -> Int {
    var fator: Int = 2
    func ret(x: Int) -> Int {
        return fator * f(x)
    }
    return ret
}
```

闭包的概念及“闭包裹携其所处环境数据”的特点，正是为了闭包自己功能完整，不管闭包被赋值到哪里都能完整可用。

闭包、lambda、匿名函数的关系

Lambdas are a language construct (lambda is an anonymous function)。 closures are an implementation technique to implement first-class functions (whether anonymous or not)

Closures take one of three form:

Global functions are closures that have a name and do not capture any values.
Nested functions are closures that have a name and can capture values from their enclosing function.
Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context.

总结一下的话：lambdas就是匿名函数，一种匿名的形式或者表达式来表示函数。闭包跟匿名不匿名没什么关系，闭包更强调一种技术实现。但最终大家还是可以混着说。个人更倾向于swift定义，普通函数，lambdas，匿名函数都是闭包，闭包具体的实现了让各种函数是一等函数的能力。

swift语言在定义函数的时候，会用一个箭头 ->来表示函数的返回值，因为普通函数也是闭包，是一等函数，用->就显得非常统一合理。

```
//数学函数
f(x, y) = x + 2y;

f(1, 2) = 5;
```

令 x = 1, 我们会得到一个新的函数 g(y):

```
g(y) = f(1, y) = 1 + 2y;
g(2) = 1 + 4 = 5;
```

```
//Swift 函数

func f(x: Int, _ y: Int) -> Int {
    return x + 2 * y;
}

f(1,2)//5
```

```
func f(x: Int) -> Int -> Int {
    func ret(y: Int) -> Int {
        return x + 2 * y
    }
    return ret
}

var g = f(1)
g(2)//5
f(1)(2)//5
```

柯里化（Currying）

在计算机科学中，**柯里化（Currying）**是把接受多个参数的函数变换成接受一个单一参数(最初函数的第一个参数)的函数，并且返回接受余下的参数且返回结果的新函数的技术。这个技术由 Christopher Strachey 以逻辑学家 Haskell Curry 命名的，尽管它是 Moses Schnfinkel 和 Gottlob Frege 发明的。

函数式编程 (functional programming)

什么是函数式编程，wiki页面的解释简单充分。

- “In computer science, functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.”

这句话说函数式编程是一种编程范式，编程范式是一种编写程序的风格，函数式编程把函数写的像数学函数那样，编程计算就像函数运算，避免使用状态改变和可变数据类型。

- In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result f(x) each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

这就话提到了进行函数式编程的一条主要动机：无副作用。数学中对函数的定义是从一个集合到另一个集合的映射，可以是多对一的映射但是不能是一对多。数学函数不会出现f(1)即等于a又等于b的情况。正好对应无副作用的概念，或者是重入概念。无副作用和重入还有其它定义。这里强调的是：函数只要有相同的输入参数总会有相同的返回值，并且举了个例子，要消除函数里面状态改变语法，因为状态改变是不受函数参数控制的变化，这样的状态改变会引起副作用。

- Imperative programming does have functions—not in the mathematical sense—but in the sense of subroutines.

这一句说函数与函数式区别:指令式编程中的函数和数学函数没多大关系，指令式编程中的函数是一种subroutines（子程序）。Imperative programming 有的地方也叫 Procedural programming （ include C, Go, Fortran, Pascal, and BASIC）。以c语言来讲，c语言函数是一系列指令的打包，编译成汇编也是对应的一片指令，和数学函数概念相差很远。另外忽略运行时和虚拟机一些技术，c语言这类函数（subroutines、procedure）加载到内存后处于代码区（text segment、code segment），此区域是只读的，是不能在运行时的产生的。这类函数编译之后就是一系列的计算机指令和操作。可见与函数式编程中的一等函数差异是非常明显的。

最后一个小总结函数式编程的特征：

- 函数是一等函数，可以像其它数据类型出现在任何地方。
- 无副作用。
- 所有都是表达式，没有`for`，`switch`等状态控制。
- 使用函数组合来表达一连串的逻辑。

λ-calculus 、 Y combinator 和递归（Recursion）

Motivating Example：

阶乘的数学递归定义：0! = 1，n! = (n-1) ! × n;

```
func factorial(n: Int) -> Int {
    return n == 0 ? 1 : n * factorial(n - 1)
}

factorial(3)//6
```

```
let factorial : Int -> Int = { (n: Int) -> Int in
    return n == 0 ? 1 : n * factorial(n - 1)
}

//Variable used within its own initial value
//如何解决？
```

λ-calculus：

- Lambda calculus provides a theoretical framework for describing functions and their evaluation. Although it is a mathematical abstraction rather than a programming language, it forms the basis of almost all functional programming languages today.【10】

Lambda calculus 由Alonzo Church提出，是一整套理论计算框架，一种数学抽象而不是具体的计算机语言。是函数式编程语言的理论基础。λ演算被证明是图灵等价。

看个最简单的Lambda calculus:
λx.x^2+2

这个lambda表达式定义了一个匿名函数，这个函数的自变量是x。
对应的函数表示f(x) = x^2 + 2；λx表示x是自变量，x^2+2是函数表达式；
f(2) = 2^2 + 2 = 6；
可以通用λ表达式表达：
(λx.x^2+2)2 //结果是6

Y Combinator就是下面这个lambda表达式：
Y ： (λy.(λx.y(xx))(λx.y(xx)))

不动点：
x = f(x)
f是一个函数映射，当给f传入参数x，经过f的映射之后返回的值还是x。我们称x是函数f的不动点。
Y combinator的作用是可以找到函数的不动点。

```
Y ： (λy.(λx.y(xx))(λx.y(xx)))  //Y Combinator
YR = (λx.R (xx))  (λx.R (xx))   //代入R,进行计算
= R ((λx.R (xx)) (λx.R (xx))

YR = R(YR)                      //可见YR是R的不动点！
```

正是Y combinator可得到一个函数的不动点的功效，让匿名函数递归成为可能。要实现匿名函数递归具体步骤如下：

Y Combinator:

```
func Y<A, B>(f: (A -> B) -> A -> B) -> A -> B {
    typealias RecursiveFunc = Any -> A -> B
    let r : RecursiveFunc = { (z: Any) in
        let w = z as! RecursiveFunc;
        return f {
            print(w.dynamicType)
            return w(w)($0)
        }
    }
    return r(r)
}
```

step 1:

```
typealias F = Int -> Int
var f:F -> F = {(f: F) -> F in
    var ret:Int -> Int = {(n: Int) -> Int in
        if (n == 0) {
            return 1;
        }
        return n * f(n - 1)
    }
    return ret;
}
```

https://nikolaygrozev.wordpress.com/2013/12/10/monads-in-15-minutes/

step 2:

```
var factorial = Y(f);
factorial(3)//6
```

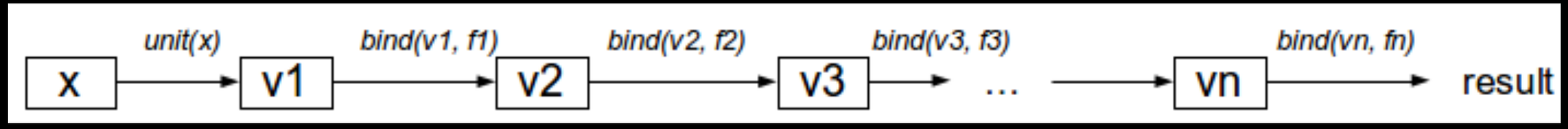
Swift & functional programming

1. `func Y<A, B>(f: (A -> B) -> A -> B) -> A -> B {}`

2. **optional types:**

“they are at the heart of many of Swift’s most powerful features.”

Monads


unit(x)
bind(v1, f1)
bind(v2, f2)
bind(v3, f3)
bind(vn, fn)
x
v1
v2
v3
...
vn
result

```swift
func payWaterBills(salary: Double) -> (Double, String) {
    if salary >= 100 {
        //pay water bills
        return (salary - 100, "")
    } else {
        return (salary, "payWaterBills error")
    }
}

func payElectricityBills(salary: Double) -> (Double, String) {
    if salary >= 300 {
        //pay electricity bills
        return (salary - 300, "")
    } else {
        return (salary, "payElectricityBills error")
    }
}

func payInternetBills(salary: Double) -> (Double, String) {
    if salary >= 500 {
        //pay internet bills
        return (salary - 500, "")
    } else {
        return (salary, "payInternetBills error")
    }
}

func tryPayBills2(salary: Double) -> (Double, String) {
    let salaryU = payWaterBills(salary)
    if salaryU.1 == "" {
        let salaryU = payElectricityBills(salaryU.0)
        if salaryU.1 == "" {
            let salaryU = payInternetBills(salaryU.0)
            return salaryU
        } else {
            return salaryU
        }
    } else {
        return salaryU
    }
}
```

```swift
print(tryPayBills2(30))
print(tryPayBills2(230))
print(tryPayBills2(630))
print(tryPayBills2(1000))
```

```
(30.0, "payWaterBills  error")
(130.0, "payElectricityBills  error")
(230.0, "payInternetBills  error")
(100.0, "")
```

```
func unit(salary: Double) -> (Double, String) {
    return (salary, "")
}

func bind(salaryU:(Double, String), _ payBills: Double ->  (Double, String)) -> (Double, Strin
    if salaryU.1 == "" {
        return payBills(salaryU.0)
    } else {
        return salaryU
    }
}
```

```
func tryPayBills1(salary: Double) -> (Double, String) {
    return bind(bind(bind(unit(salary), payWaterBills), payElectricityBills), payInternetBills
}



print(tryPayBills1(30))
print(tryPayBills1(230))
print(tryPayBills1(630))
print(tryPayBills1(1000))



(30.0, "payWaterBills error")
(130.0, "payElectricityBills error")
(230.0, "payInternetBills error")
(100.0, "")
```

Optional Type:

var salary: Double? = Double("11")

Optional Binding: if let

```
if let salaryNum = salary {
    print("my salary \(salaryNum)")
} else {
    print("something wrong.")
}
```

Optional Chaining:

```
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
    print("John's building identifier is \(buildingIdentifier).")
}
// Prints "John's building identifier is The Larches."
```

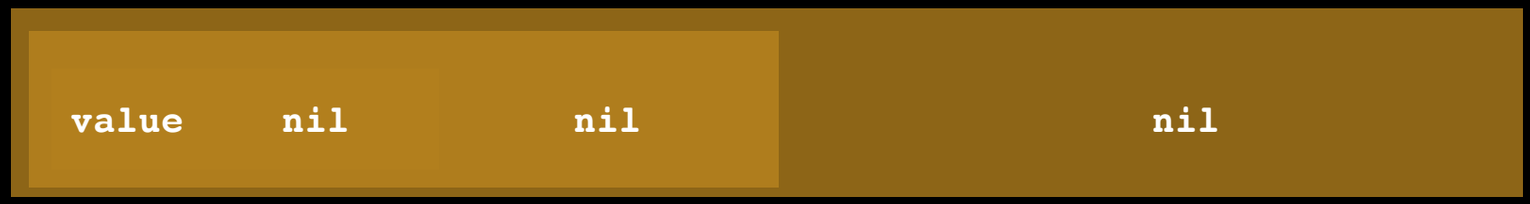
nil:

Swift’s nil is not the same as nil in Objective-C. In Objective-C, nil is a pointer to a nonexistent object. In Swift, nil is not a pointer—it is the absence of a value of a certain type. Optionals of *any* type can be set to nil, not just object types.

多个角度理解：**Optional / if let**

```
var x: String???
```

//Monads, enumeration




```
x = nil;
x = Optional.None;
x = Optional.Some(Optional.None);
x = Optional.Some(Optional.Some((Optional.None)));
x = Optional.Some(Optional.Some((Optional.Some("value"))));
x = "value";

 if let s = x {
    print(" \(x)")
    print(" \(s)")

} else {
    print("there is no s !")
}


if x != nil {
    let s = x!
    print(" \(x)")
    print(" \(s)")

} else {
    print("there is no s !")
}
```

Thank You