

函数的定义



扫码试看/订阅

《Swift核心技术与实战》视频课程

基本概念

- 函数是一个独立的代码块，用来执行特定的任务。通过给函数一个名字来定义它的功能，并且在需要的时候，通过这个名字来“调用”函数执行它的任务
- Swift 统一的函数语法十分灵活，可以表达从简单的无形式参数的 C 风格函数到复杂的每一个形式参数都带有局部和外部形式参数名的 Objective-C 风格方法的任何内容。形式参数能提供一个默认的值来简化函数的调用，也可以被当作输入输出形式参数被传递，它在函数执行完成时修改传递来的变量。
- Swift 中的每一个函数都有类型，由函数的形式参数类型和返回类型组成。你可以像 Swift 中其他类型那样来使用它，这使得你能够方便的将一个函数当作一个形式参数传递到另外的一个函数中，也可以在一个函数中返回另一个函数。函数同时也可以写在其他函数内部来在内嵌范围封装有用的功能。

定义和调用函数

- 当你定义了一个函数的时候，你可以选择定义一个或者多个命名的分类的值作为函数的输入（所谓的形式参数），并且/或者定义函数完成后将要传回作为输出的值的类型（所谓它的返回类型）
- 每一个函数都有一个函数名，它描述函数执行的任务。要使用一个函数，你可以通过“调用”函数的名字并且传入一个符合函数形式参数类型的输入值（所谓实际参数）来调用这个函数。给函数提供的实际参数的顺序必须符合函数的形式参数列表顺序。

```
func greet(person: String) -> String {  
    let greeting = "Hello, " + person + "!"  
    return greeting  
}
```

无形式参数的函数

- 函数没有要求必须输入一个参数，可以没有形式参数。
- 函数的定义仍然需要在名字后边加一个圆括号，即使它不接受形式参数也得这样做。当函数被调用的时候也要在函数的名字后边加一个空的圆括号。

```
func sayHelloWorld() -> String {  
    return "hello, world"  
}  
print(sayHelloWorld())  
// prints "hello, world"
```

多形式参数的函数

- 函数可以输入多个形式参数，可以写在函数后边的圆括号内，用逗号分隔。

```
func greet(person: String, alreadyGreeted: Bool) -> String {  
    if alreadyGreeted {  
        return greetAgain(person: person)  
    } else {  
        return greet(person: person)  
    }  
}  
  
print(greet(person: "Tim", alreadyGreeted: true))  
// Prints "Hello again, Tim!"
```

无返回值的函数

- 函数定义中没有要求必须有一个返回类型。
- 不需要返回值，函数在定义的时候就没有包含返回箭头（->）或者返回类型。
- 严格来讲，函数 `greet(person:)` 还是有一个返回值的，尽管没有定义返回值。没有定义返回类型的函数实际上会返回一个特殊的类型 `Void`。它其实是一个空的元组，作用相当于没有元素的元组，可以写作 `()`。

```
func greet(person: String) {  
    print("Hello, \(person)!")  
}  
greet(person: "Dave")
```


多返回值的函数

- 为了让函数返回多个值作为一个复合的返回值，你可以使用元组类型作为返回类型。

```
func minMax(array: [Int]) -> (min: Int, max: Int) {  
    var currentMin = array[0]  
    var currentMax = array[0]  
    for value in array[1..<array.count] {  
        if value < currentMin {  
            currentMin = value  
        } else if value > currentMax {  
            currentMax = value  
        }  
    }  
    return (currentMin, currentMax)  
}
```


可选元组返回类型

- 如果元组在函数的返回类型中有可能“没有值”，你可以用一个可选元组返回类型来说明整个元组的可能为 nil。写法是在可选元组类型的圆括号后边添加一个问号（?）例如 (Int, Int)? 或者 (String, Int, Bool)?。

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {  
    if array.isEmpty { return nil }  
    var currentMin = array[0]  
    var currentMax = array[0]  
    for value in array[1..        if value < currentMin {  
            currentMin = value  
        } else if value > currentMax {  
            currentMax = value  
        }  
    }  
    return (currentMin, currentMax)  
}
```

```
if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {  
    print("min is \(bounds.min) and max is \(bounds.max)")  
}
```

隐式返回的函数

- 如果整个函数体是一个单一表达式，那么函数隐式返回这个表达式。

```
func greeting(for person: String) -> String {  
    "Hello, " + person + "!"  
}
```

```
print(greeting(for: "Dave"))  
// Prints "Hello, Dave!"
```

```
func anotherGreeting(for person: String) -> String {  
    return "Hello, " + person + "!"  
}
```

```
print(anotherGreeting(for: "Dave"))  
// Prints "Hello, Dave!"
```



函数实际参数标签和形式参数名

实参标签和形参名

- 每一个函数的形式参数都包含实际参数标签和形式参数名。实际参数标签用在调用函数的时候；在调用函数的时候每一个实际参数前边都要写实际参数标签。形式参数名用在函数的实现当中。默认情况下，形式参数使用它们的形式参数名作为实际参数标签。
- 所有的形式参数必须有唯一的名字。尽管有可能多个形式参数拥有相同的实际参数标签，唯一的实际参数标签有助于让你的代码更加易读。

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {  
    // In the function body, firstParameterName and secondParameterName  
    // refer to the argument values for the first and second parameters.  
}  
someFunction(firstParameterName: 1, secondParameterName: 2)
```

指定实际参数标签

- 在提供形式参数名之前写实际参数标签，用空格分隔。
- 如果你为一个形式参数提供了实际参数标签，那么这个实际参数就必须在调用函数的时候使用标签。
- 实际参数标签的使用能够让函数的调用更加明确，更像是自然语句，同时还能提供更可读的函数体并更清晰地表达你的意图。


```
func greet(person: String, from hometown: String) -> String {  
    return "Hello \$(person)! Glad you could visit from \$(hometown)."  
}  
  
print(greet(person: "Bill", from: "Cupertino"))  
  
// Prints "Hello Bill! Glad you could visit from Cupertino."
```


省略实际参数标签

- 如果对于函数的形式参数不想使用实际参数标签的话，可以利用下划线（`_`）来为这个形式参数代替显式的实际参数标签。

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {  
    // In the function body, firstParameterName and secondParameterName  
    // refer to the argument values for the first and second parameters.  
}  
someFunction(1, secondParameterName: 2)
```

默认形式参数值

- 你可以通过在形式参数类型后给形式参数赋一个值来给函数的任意形式参数定义一个默认值。
- 如果定义了默认值，你就可以在调用函数时候省略这个形式参数。 

```
func someFunction(parameterWithDefault: Int = 12) {  
    // In the function body, if no arguments are passed to the function  
    // call, the value of parameterWithDefault is 12.  
}  
someFunction(parameterWithDefault: 6) // parameterWithDefault is 6  
someFunction() // parameterWithDefault is 12
```


可变形式参数

- 一个可变形式参数可以接受零或者多个特定类型的值。当调用函数的时候你可以利用可变形式参数来声明形式参数可以被传入值的数量是可变的。可以通过在形式参数的类型名称后边插入三个点符号（...）来书写可变形式参数。
- 传入到可变参数中的值在函数的主体中被当作是对应类型的数组。

```
func arithmeticMean(_ numbers: Double...) -> Double {  
    var total: Double = 0  
    for number in numbers {  
        total += number  
    }  
    return total / Double(numbers.count)  
}  
arithmeticMean(1, 2, 3, 4, 5)  
// returns 3.0, which is the arithmetic mean of these five numbers  
arithmeticMean(3, 8.25, 18.75)  
// returns 10.0, which is the arithmetic mean of these three numbers
```

输入输出形式参数

- 可变形式参数只能在函数的内部做改变。如果你想函数能够修改一个形式参数的值，而且你想这些改变在函数结束之后依然生效，那么就需要将形式参数定义为输入输出形式参数。
- 在形式参数定义开始的时候在前边添加一个 inout 关键字可以定义一个输入输出形式参数。输入输出形式参数有一个能输入给函数的值，函数能对其进行修改，还能输出到函数外边替换原来的值。
- 你只能把变量作为输入输出形式参数的实际参数，在将变量作为实际参数传递给输入输出形式参数的时候，直接在它前边添加一个和符号 (&) 来明确可以被函数修改
- 输入输出形式参数不能有默认值，可变形式参数不能标记为 inout

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

```
var someInt = 3  
var anotherInt = 107  
swapTwoInts(&someInt, &anotherInt)  
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
```

函数类型和内嵌函数

函数类型

- 每一个函数都有一个特定的函数类型，它由形式参数类型，返回类型组成。

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {  
    return a + b  
}
```

```
func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {  
    return a * b  
}
```

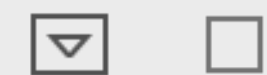


`(Int, Int) -> Int`

使用函数类型

- 你可以像使用 Swift 中的其他类型一样使用函数类型。例如，你可以给一个常量或变量定义一个函数类型，并且为变量指定一个相应的函数。

```
5 func addTwoNumber(num: Int, num2: Int) -> Int {  
6     return num + num2  
7 }  
8  
9 var mathFunction: (Int, Int) -> Int = addTwoNumber  
10 print(mathFunction(2, 3))
```



5

```
4  
5 func addTwoNumber(first num: Int, second num2: Int) -> Int {  
6     return num + num2  
7 }  
8  
9 var mathFunction: (Int, Int) -> Int = addTwoNumber  
10 print(mathFunction(2, 3))
```



5

函数类型作为形式参数类型

- 你可以利用使用一个函数的类型例如 `(Int, Int) -> Int` 作为其他函数的形式参数类型。这允许你预留函数的部分实现从而让函数的调用者在调用函数的时候提供

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {  
    print("Result: \(mathFunction(a, b))")  
}  
printMathResult(addTwoInts, 3, 5)
```


函数类型作为返回类型

- 你可以利用函数的类型作为另一个函数的返回类型。写法是在函数的返回箭头（->）后立即写一个完整的函数类型。

```
func stepForward(_ input: Int) -> Int {
    return input + 1
}

func stepBackward(_ input: Int) -> Int {
    return input - 1
}

func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    return backwards ? stepBackward : stepForward
}
```

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)

print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
```


内嵌函数

- 可以在函数的内部定义另外一个函数。这就是内嵌函数。
- 内嵌函数在默认情况下在外部是被隐藏起来的，但却仍然可以通过包裹它们的函数来调用它们。包裹的函数也可以返回它内部的一个内嵌函数来在另外的范围里使用。

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {  
    func stepForward(input: Int) -> Int { return input + 1 }  
    func stepBackward(input: Int) -> Int { return input - 1 }  
    return backward ? stepBackward : stepForward  
}  
  
var currentValue = -4  
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)  
// moveNearerToZero now refers to the nested stepForward() function  
while currentValue != 0 {  
    print("\(currentValue)... ")  
    currentValue = moveNearerToZero(currentValue)  
}  
print("zero!")
```

闭包

闭包的概念

- 闭包是可以在你的代码中被传递和引用的功能性独立代码块。
- 闭包能够捕获和存储定义在其上下文中的任何常量和变量的引用，这也就是所谓的闭合并包裹那些常量和变量，因此被称为“闭包”，Swift 能够为你处理所有关于捕获的内存管理的操作。

闭包的概念

- 在函数章节中有介绍的全局和内嵌函数，实际上是特殊的闭包。闭包符合如下三种形式中的一种：
 1. 全局函数是一个有名字但不会捕获任何值的闭包；
 2. 内嵌函数是一个有名字且能从其上层函数捕获值的闭包；
 3. 闭包表达式是一个轻量级语法所写的可以捕获其上下文中常量或变量值的没有名字的闭包。

闭包表达式

- 闭包表达式是一种在简短行内就能写完闭包的语法。

闭包表达式-从 sorted 函数说起

- Swift 的标准库提供了一个叫做 `sorted(by:)` 的方法，会根据你提供的排序闭包将已知类型的数组的值进行排序。一旦它排序完成，`sorted(by:)` 方法会返回与原数组类型大小完全相同的一个新数组，该数组的元素是已排序好的。原始数组不会被 `sorted(by:)` 方法修改。

```
13 let names = ["zhangsan", "lisi", "wangwu", "zhaoliu"]
14 func backward(_ s1: String, _ s2: String) -> Bool {
15     return s1 > s2
16 }
17 var reversedNames = names.sorted(by: backward)
18 print(reversedNames)
```



`["zhaoliu", "zhangsan", "wangwu", "lisi"]`

闭包表达式语法



- 闭包表达式语法能够使用常量形式参数、变量形式参数和输入输出形式参数，但不能提供默认值。可变形式参数也能使用，但需要在形式参数列表的最后面使用。元组也可被用来作为形式参数和返回类型。

```
{ (parameters) -> (return type) in  
  statements  
}
```


闭包表达式语法版本的 backward

- 将之前 backward(_:_:) 函数改为闭包表达版本

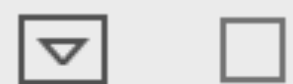
```
12
13 let names = ["zhangsan", "lisi", "wangwu", "zhaoliu"]
14 var reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
15     return s1 > s2
16 })
17 print(reversedNames)
```


 
["zhaoliu", "zhangsan", "wangwu", "lisi"]

从语境中推断类型

- 因排序闭包为实际参数来传递给函数，故 Swift 能推断它的形式参数类型和返回类型
- sorted(by:) 方法期望它的形式参数是一个 (String, String) -> Bool 类型的函数。这意味着 (String, String) 和 Bool 类型不需要被写成闭包表达式定义中的一部分，因为所有的类型都能被推断，返回箭头 (->) 和围绕在形式参数名周围的括号也能被省略

```
12
13 let names = ["zhangsan", "lisi", "wangwu", "zhaoliu"]
14 var reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
15 print(reversedNames)
```

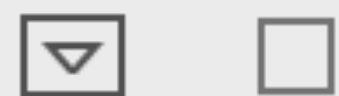


```
["zhaoliu", "zhangsan", "wangwu", "lisi"]
```

从单表达式闭包隐式返回

- 单表达式闭包能够通过从它们的声明中删掉 `return` 关键字来隐式返回它们单个表达式的结果。

```
12  
13 let names = ["zhangsan", "lisi", "wangwu", "zhaoliu"]  
14 var reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )  
15 print(reversedNames)
```



`["zhaoliu", "zhangsan", "wangwu", "lisi"]`

简写实际参数名

- Swift 自动对行内闭包提供简写实际参数名，可以通过 \$0, \$1, \$2 等名字来引用闭包的参数值。

```
13 let names = ["zhangsan", "lisi", "wangwu", "zhaoliu"]
14 var reversedNames = names.sorted(by: { $0 > $1 } )|
15 print(reversedNames)
```



```
["zhaoliu", "zhangsan", "wangwu", "lisi"]
```

运算符函数

- Swift 的 String 类型定义了关于大于号 (>) 的特定字符串实现, 让其作为一个有两个 String 类型形式参数的函数并返回一个 Bool 类型的值。这正好与 sorted(by:) 方法的形式参数需要的函数相匹配。因此, 你能简单地传递一个大于号, 并且 Swift 将推断你想使用大于号特殊字符串函数实现

```
12  
13 let names = ["zhangsan", "lisi", "wangwu", "zhaoliu"]  
14 var reversedNames = names.sorted(by: > )  
15 print(reversedNames)
```



```
["zhaoliu", "zhangsan", "wangwu", "lisi"]
```

尾随闭包

- 如果你需要将一个很长的闭包表达式作为函数最后一个实际参数传递给函数，使用尾随闭包将增强函数的可读性。尾随闭包是一个被书写在函数形式参数的括号外面（后面）的闭包表达式。

```
81 let names = ["zhangsan", "lisi", "wangwu", "zhaoliu"]
82 let reversedNames = names.sorted{ $0 > $1 }
83 print(reversedNames)
```

["zhaoliu", "zhangsan", "wangwu", "lisi"]

捕获值

捕获值

- 一个闭包能够从上下文捕获已被定义的常量和变量。即使定义这些常量和变量的原作用域已经不存在，闭包仍能够在其函数体内引用和修改这些值。

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {  
    var runningTotal = 0  
    func incrementer() -> Int {  
        runningTotal += amount  
        return runningTotal  
    }  
    return incrementer  
}
```

捕获值

- 作为一种优化，如果一个值没有改变或者在闭包的外面，Swift 可能会使用这个值的拷贝而不是捕获。
- Swift也处理了变量的内存管理操作，当变量不再需要时会被释放。

```
let incrementByTen = makeIncrementer(forIncrement: 10)
incrementByTen()
incrementByTen()
incrementByTen()
```

```
() -> Int
10
20
30
```

捕获值

- 如果你建立了第二个 incrementer ,它将会有一个新的、独立的 runningTotal 变量的引用。

```
let incrementByTen = makeIncrementer(forIncrement: 10)
incrementByTen()
incrementByTen()
incrementByTen()

let incrementBySeven = makeIncrementer(forIncrement: 7)
incrementBySeven()

incrementByTen()
```

() -> Int
10
20
30
() -> Int
7
40

闭包是引用类型

- 在 Swift 中，函数和闭包都是引用类型。
- 无论你在什么时候赋值一个函数或者闭包给常量或者变量，你实际上都是将常量和变量设置为对函数和闭包的引用。

<code>let incrementByTen = makeIncrementer(forIncrement: 10)</code>	<code>() -> Int</code>
<code>incrementByTen()</code>	10
<code>incrementByTen()</code>	20
<code>incrementByTen()</code>	30
<code>let incrementBySeven = makeIncrementer(forIncrement: 7)</code>	<code>() -> Int</code>
<code>incrementBySeven()</code>	7
<code>incrementByTen()</code>	40
<code>let alsoIncrementByTen = incrementByTen</code>	<code>() -> Int</code>
<code>alsoIncrementByTen()</code>	50

闭包是引用类型

- 如果你分配了一个闭包给类实例的属性，并且闭包通过引用该实例或者它的成员来捕获实例，你将在闭包和实例间会产生循环引用。

逃逸闭包和自动闭包

逃逸闭包

- 当闭包作为一个实际参数传递给一个函数的时候，并且它会在函数返回之后调用，我们就说这个闭包逃逸了。当你声明一个接受闭包作为形式参数的函数时，你可以在形式参数前写 `@escaping` 来明确闭包是允许逃逸的。
- 闭包可以逃逸的一种方法是被储存在定义于函数外的变量里。比如说，很多函数接收闭包实际参数来作为启动异步任务的回调。函数在启动任务后返回，但是闭包要直到任务完成——闭包需要逃逸，以便于稍后调用。

```
var completionHandlers: [() -> Void] = []  
func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void) {  
    completionHandlers.append(completionHandler)  
}
```

逃逸闭包

- 让闭包 @escaping 意味着你必须在闭包中显式地引用 self

```
func someFunctionWithNonescapingClosure(closure: () -> Void) {  
    closure()  
}
```

```
class SomeClass {  
    var x = 10  
    func doSomething() {  
        someFunctionWithEscapingClosure { self.x = 100 }  
        someFunctionWithNonescapingClosure { x = 200 }  
    }  
}
```

```
let instance = SomeClass()  
instance.doSomething()  
print(instance.x)  
// Prints "200"
```

```
completionHandlers.first?()  
print(instance.x)
```

自动闭包

- 自动闭包是一种自动创建的用来把作为实际参数传递给函数的表达式打包的闭包。它不接受任何实际参数，并且当它被调用时，它会返回内部打包的表达式的值。
- 这个语法的好处在于通过写普通表达式代替显式闭包而使你省略包围函数形式参数的括号。

```
public func assert(_ condition: @autoclosure () -> Bool, _ message: @autoclosure () -  
> String = String(), file: StaticString = #file, line: UInt = #line)
```

```
let number = 3  
assert(number > 3, "number 不大于3")
```

自动闭包

- 自动闭包允许你延迟处理，因此闭包内部的代码直到你调用它的时候才会运行。对于有副作用或者占用资源的代码来说很有用，因为它可以允许你控制代码何时才进行求值。

```
var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
print(customersInLine.count)
// Prints "5"

let customerProvider = { customersInLine.remove(at: 0) }
print(customersInLine.count)
// Prints "5"

print("Now serving \(customerProvider())!")
// Prints "Now serving Chris!"
print(customersInLine.count)
// Prints "4"
```

自动闭包

- 当你传一个闭包作为实际参数到函数的时候，你会得到与延迟处理相同的行为。

```
// customersInLine is ["Alex", "Ewa", "Barry", "Daniella"]  
func serve(customer customerProvider: () -> String) {  
    print("Now serving \(customerProvider())!")  
}  
  
serve(customer: { customersInLine.remove(at: 0) } )  
// Prints "Now serving Alex!"
```


自动闭包

- 通过 @autoclosure 标志标记它的形式参数使用了自动闭包。现在你可以调用函数就像它接收了一个 String 实际参数而不是闭包。实际参数自动地转换为闭包，因为 customerProvider 形式参数的类型被标记为 @autoclosure 标记。

```
// customersInLine is ["Ewa", "Barry", "Daniella"]  
func serve(customer customerProvider: @autoclosure () -> String) {  
    print("Now serving \(customerProvider())!")  
}  
  
serve(customer: customersInLine.remove(at: 0))  
// Prints "Now serving Ewa!"
```


自动+逃逸

- 如果你想要自动闭包允许逃逸，就同时使用 @autoclosure 和 @escaping 标志。

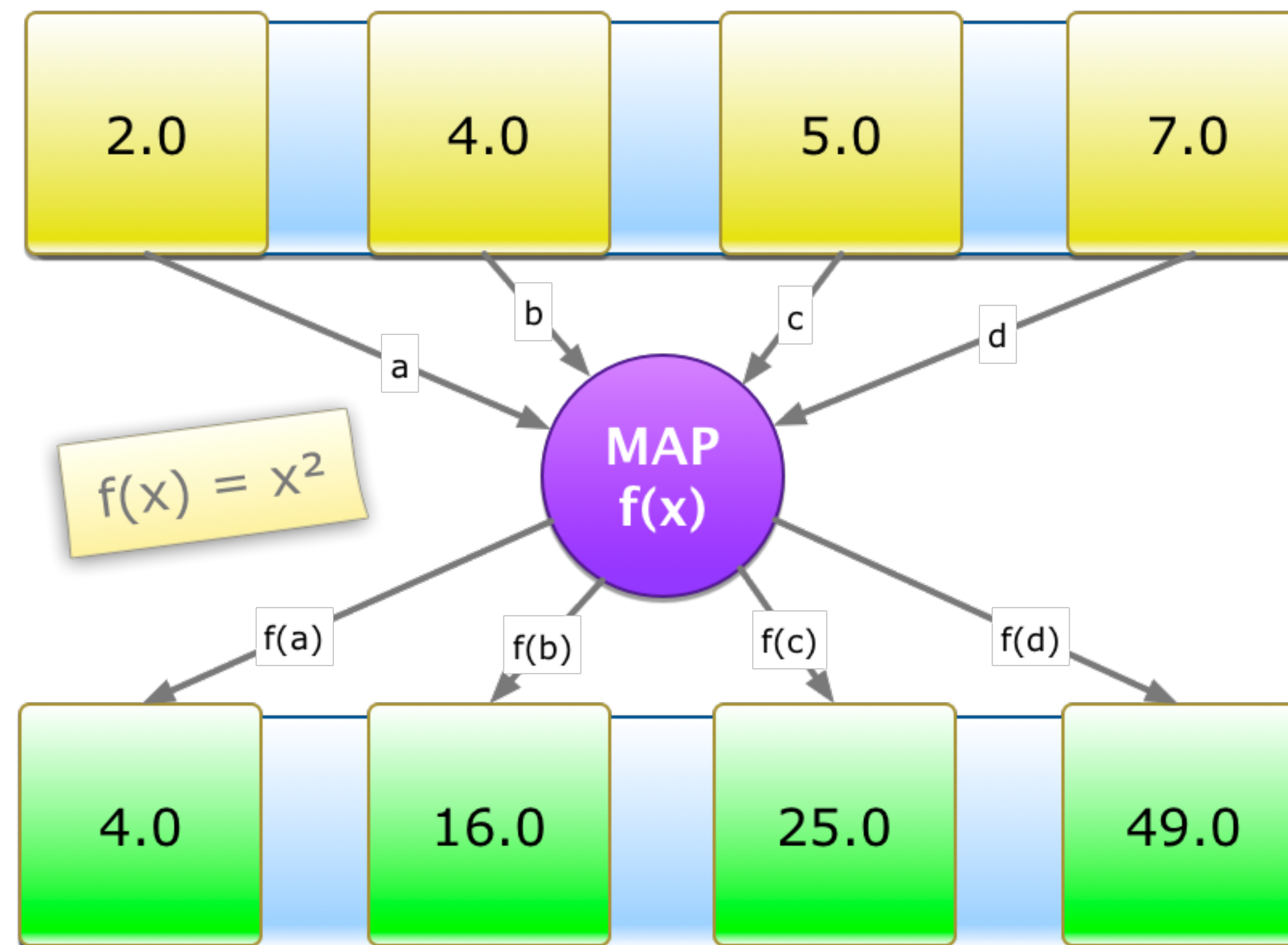
```
// customersInLine is ["Barry", "Daniella"]
var customerProviders: [() -> String] = []
func collectCustomerProviders(_ customerProvider: @autoclosure @escaping () -> String) {
    customerProviders.append(customerProvider)
}
collectCustomerProviders(customersInLine.remove(at: 0))
collectCustomerProviders(customersInLine.remove(at: 0))

print("Collected \(customerProviders.count) closures.")
// Prints "Collected 2 closures."
for customerProvider in customerProviders {
    print("Now serving \(customerProvider())!")
}
// Prints "Now serving Barry!"
// Prints "Now serving Daniella!"
```

高阶函数

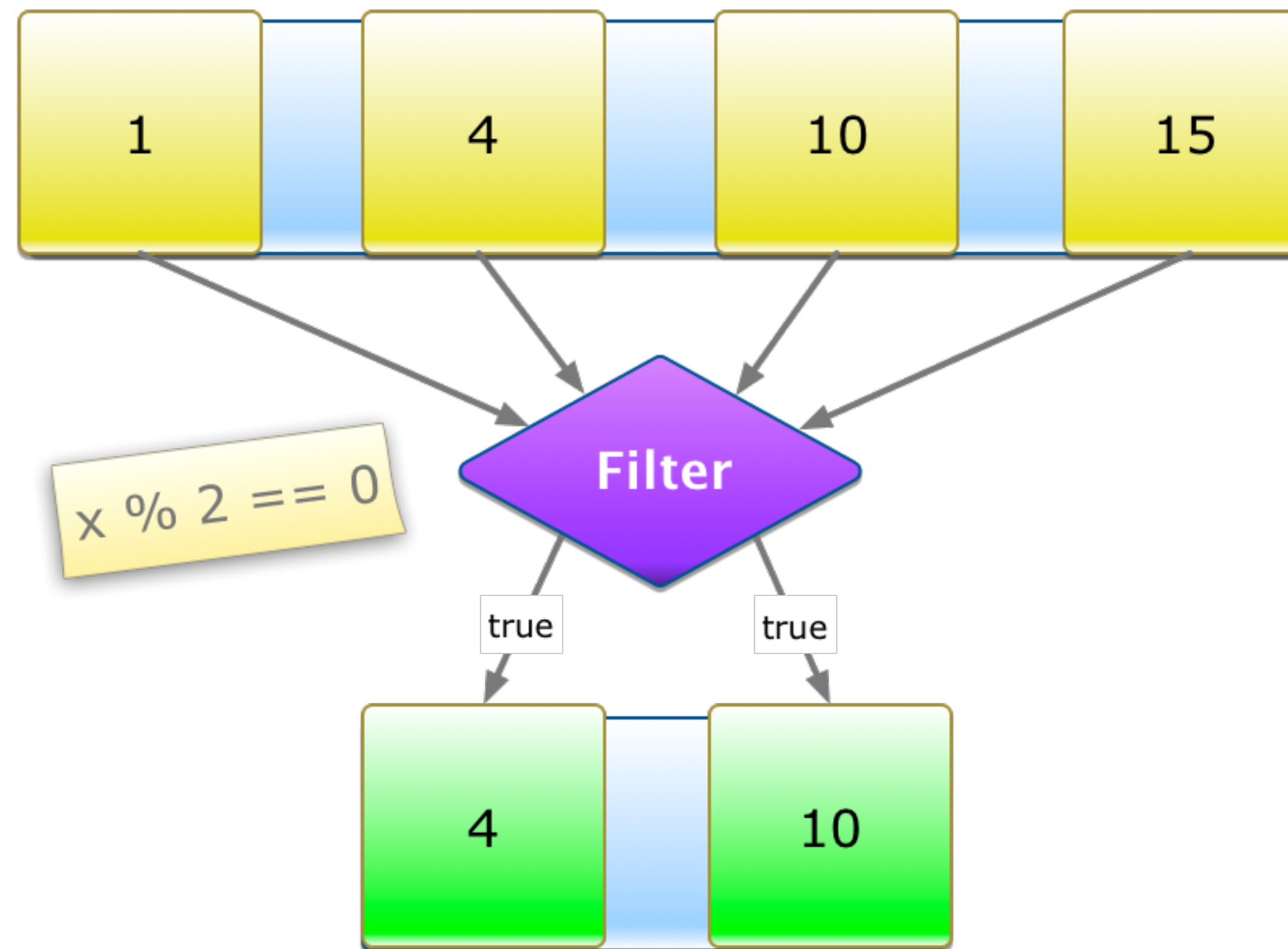
map

- 对于原始集合里的每一个元素，以一个变换后的元素替换之形成一个新的集合



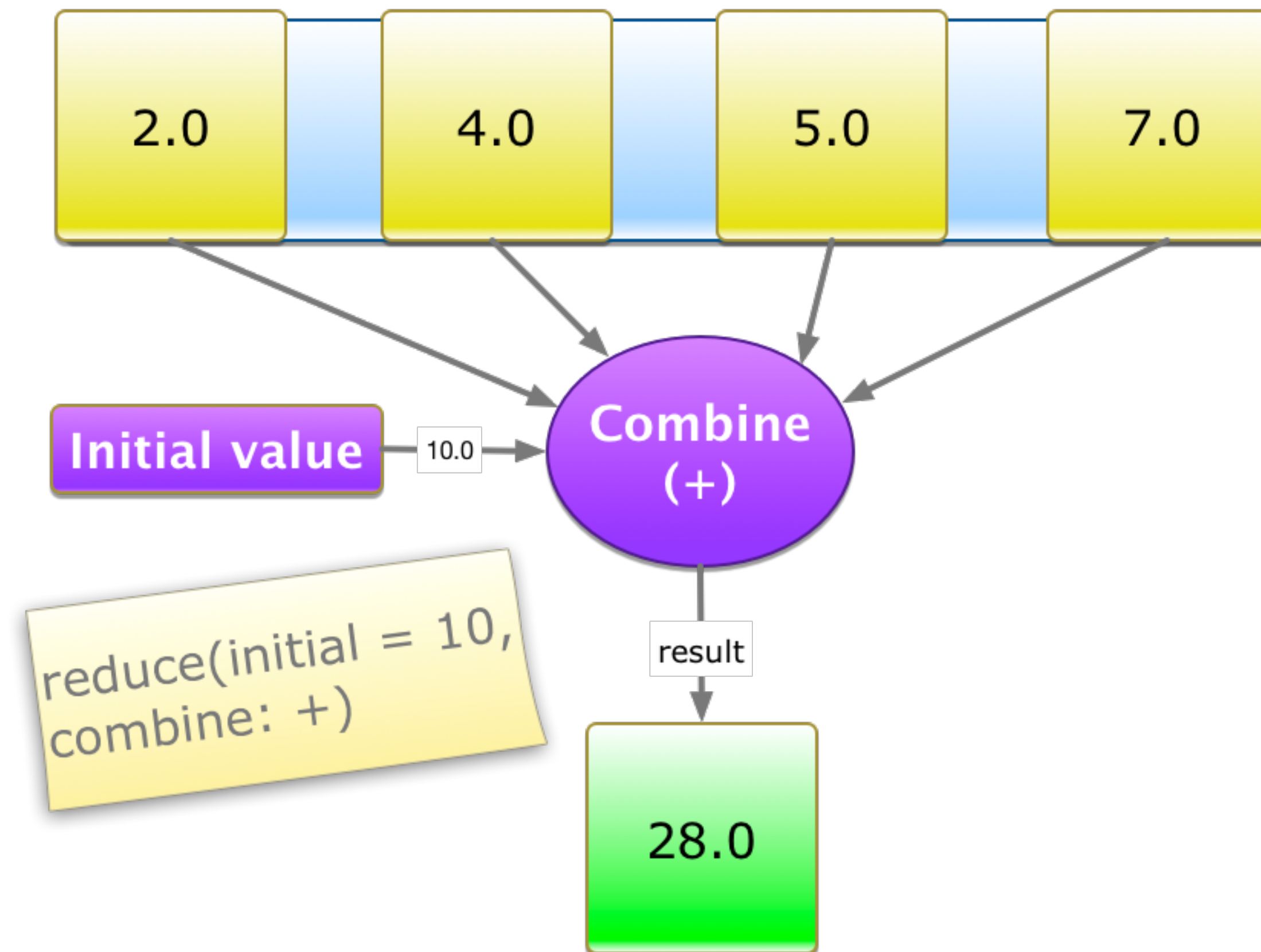
filter

- 对于原始集合里的每一个元素，通过判定来将其丢弃或者放进新集合



reduce

- 对于原始集合里的每一个元素，作用于当前累积的结果上






flatMap

还有joined

- 对于元素是集合的集合，可以得到单级的集合

```
105 let results = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
106 let allResults = results.flatMap { $0.map { $0 * 10 } }
107 let passMarks = results.flatMap { $0.filter { $0 > 5} }
108 print(allResults)
109 print(passMarks)
```


 
[10, 20, 30, 40, 50, 60, 70, 80, 90]
[6, 7, 8, 9]

compactMap

- 过滤空值

```
111
112 let keys: [String?] = ["Zhangsan", nil, "Lisi", nil, "Wangwu"]
113 let validNames = keys.compactMap { $0 }
114 print(validNames)
115 let counts = keys.compactMap { $0?.count }
116 print(counts)
```



```
["Zhangsan", "Lisi", "Wangwu"]
[8, 4, 6]
```

函数式编程1

范式转换-从一个题目说起

- 读入一个文本文件，确定所有单词的使用频率并从高到低排序，打印出所有单词及其频率的排序列表。
- 这道题目出自计算机科学史上的著名事件，是当年 Communications of the ACM 杂志 “Programming Pearls” 专栏的作者 Jon Bentley 向计算机科学先驱 Donald Knuth 提出的挑战

范式转换-传统解决方案

```
5 let NON_WORDS: Set = ["the", "and", "of", "to",  
    "a", "i", "it", "in", "or", "is", "as", "so",  
    "but", "be"]  
  
6  
7  
8 func wordFreq(words: String) -> [String: Int] {  
9     var wordDict: [String: Int] = [:]  
10    let wordList = words.split(separator: " ")  
11    for word in wordList {  
12        let lowerCaseWord = word.lowercased()  
13        if !NON_WORDS.contains(lowerCaseWord) {  
14            if let count = wordDict[lowerCaseWord] {  
15                wordDict[lowerCaseWord] = count + 1  
16            } else {  
17                wordDict[lowerCaseWord] = 1  
18            }  
19        }  
20    }  
21    return wordDict;  
22 }
```

```
25 let words = ""  
26 There are moments in life when you miss someone so  
    much that you just want to pick them from your  
    dreams and hug them for real Dream what you  
    want to dream go where you want to go be what  
    you want to be because you have only one life  
    and one chance to do all the things you want to  
    do  
27 ""  
28 print(wordFreq(words: words))
```

```
["dream": 2, "your": 1, "life": 2, "want": 5, "that": 1, "d  
1, "real": 1, "when": 1, "hug": 1, "are": 1, "for": 1, "go"  
"someone": 1, "from": 1, "there": 1, "you": 7, "because": 1
```

范式转换-函数式

```
func wordFreq2(words: String) -> [String: Int] {  
    var wordDict: [String: Int] = [:]  
    let wordList = words.split(separator: " ")  
    wordList.map { $0.lowercased() }  
        .filter{ !NON_WORDS.contains($0) }  
        .forEach { (word) in  
            wordDict[word] = (wordDict[word] ?? 0) + 1  
        }  
    return wordDict;  
}
```

```
37  
38 let words = ""  
39 There are moments in life when you miss someone so much  
    that you just want to pick them from your dreams and  
    hug them for real Dream what you want to dream go  
    where you want to go be what you want to be because  
    you have only one life and one chance to do all the  
    things you want to do
```

```
40 ""  
41 print(wordFreq2(words: words))
```

```
["pick": 1, "because": 1, "from": 1, "all": 1, "just": 1, "your":  
"for": 1, "miss": 1, "only": 1, "much": 1, "hug": 1, "where": 1, "  
"chance": 1, "there": 1, "someone": 1, "are": 1, "dreams": 1, "one
```


范式转换

- 命令式编程风格常常迫使我们出于性能考虑，把不同的任务交织起来，以便能够用一次循环来完成多个任务。
- 而函数式编程用 `map()`、`filter()` 这些高阶函数把我们解放出来，让我们站在更高的抽象层次上去考虑问题，把问题看得更清楚。

简洁

- 面向对象编程通过封装不确定因素来使代码能被人理解；函数式编程通过尽量减少不确定因素来使代码能被人理解。

简洁

- 在面向对象的命令式编程语言里面，重用的单元是类和类之间沟通用的消息。
- 函数式编程语言实现重用的思路很不一样。函数式语言提倡在有限的几种关键数据结构（如 `list`、`set`、`map`）上运用针对这些数据结构高度优化过的操作，以此构成基本的运转机构。开发者再根据具体用途，插入自己的数据结构和高阶函数去调整机构的运转方式。

简洁 j

- 比起一味创建新的类结构体系，把封装的单元降低到函数级别，更有利于达到细粒度的、基础层面的重用。
- 函数式程序员喜欢用少数几个核心数据结构，围绕它们去建立一套充分优化的运转机构。面向对象程序员喜欢不断地创建新的数据结构和附属的操作，因为压倒一切的面向对象编程范式就是建立新的类和类间的消息。把所有的数据结构都封装成类，一方面压制了方法层面的重用，另一方面鼓励了大粒度的框架式的重用。函数式编程的程序构造更方便我们在比较细小的层面上重用代码。

课后题

- 找到一个字符串里面某个字符数组里面第一个出现的字符的位置。比如 “Hello, World” , [“a” , “e” , “i” , “o” , “u”], 那 e 是在字符串第一个出现的字符, 位置是 1, 返回 1
- 提示: zip函数

函数式编程2

业务需求

- 假设我们有一个名字列表，其中一些条目由单个字符构成。现在的任务是，将除去单字符条目之外的列表内容，放在一个逗号分隔的字符串里返回，且每个名字的首字母都要大写。

命令式解法

- 命令式编程是按照“程序是一系列改变状态的命令”来建模的一种编程风格。传统的 for 循环是命令式风格的绝好例子：先确立初始状态，然后每次迭代都执行循环体中的一系列命令。

命令式解法

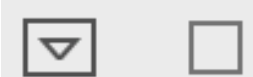
```
let employee = ["neal", "s", "stu", "j", "rich", "bob", "aiden", "j", "ethan",  
               "liam", "mason", "noah", "lucas", "jacob", "jack"]  
  
func cleanNames(names: Array<String>) -> String {  
    var cleanedNames = ""  
    for name in names {  
        if name.count > 1 {  
            cleanedNames += name.capitalized + ","  
        }  
    }  
    cleanedNames.remove(at: cleanedNames.index(before: cleanedNames.endIndex))  
    return cleanedNames  
}  
  
print(cleanNames(names: employee))
```

函数式解法

- 函数式编程将程序描述为表达式和变换，以数学方程的形式建立模型，并且尽量避免可变的状态。函数式编程语言对问题的归类不同于命令式语言。如前面所用到的几种操作（filter、transform、convert），每一种都作为一个逻辑分类由不同的函数所代表，这些函数实现了低层次的变换，但依赖于开发者定义的高阶函数作为参数来调整其低层次运转机构的运作。

函数式解法 joined ✓

```
74
75 let employee = ["neal", "s", "stu", "j", "rich", "bob",
    "aiden", "j", "ethan", "liam", "mason", "noah",
    "lucas", "jacob", "jack"]
76
77 let cleanedNames = employee.filter{ $0.count > 1 }
78                             .map{ $0.capitalized }
79                             .joined(separator: ",")
80
81
82 print(cleanedNames)
```



Neal, Stu, Rich, Bob, Aiden, Ethan, Liam, Mason, Noah, Lucas, Jacob, Jack

聊聊 Swift 的劣势-并行

```
public String cleanNamesP(List<String> names) {  
    if (names == null) return "";  
    return names  
        .parallelStream()  
        .filter(n -> n.length() > 1)  
        .map(e -> capitalize(e))  
        .collect(Collectors.joining(","));  
}
```

对 Swift 的尝试改进

```
extension Array where Element: Any {  
    func parallelMap<T>(_ transform: (Element) -> T) -> [T] {  
  
        let n = self.count  
        if n == 0 {  
            return []  
        }  
  
        var result = ContiguousArray<T>()  
        result.reserveCapacity(n)  
  
        DispatchQueue.concurrentPerform(iterations: n) { (i) in  
            result.append(transform(self[i]))  
        }  
  
        return Array<T>(result)  
    }  
}
```


对 Swift 的尝试改进

```
95 let employee = ["neal", "s", "stu", "j", "rich", "bob", "aiden",  
    "j", "ethan", "liam", "mason", "noah", "lucas", "jacob",  
    "jack"]  
96  
97 let cleanedNames = employee.filter{ $0.count > 1 }  
98     .parallelMap{ $0.capitalized }  
99     .joined(separator: ",")  
100  
101  
102 print(cleanedNames)
```



Mason, Lucas, Jack, Aiden, Noah, Stu, Jacob, Bob, Rich, Neal, Ethan, Liam

课后练习

- parallelMap 是非线程安全的，尝试改为线程安全

具有普遍意义的基本构造单元

- 筛选 (filter)
- 映射 (map)
- 折叠/化约 (foldLeft/reduce等)



扫码试看/订阅

《Swift核心技术与实战》视频课程