函数式编程

@M了个J

https://github.com/CoderMJLee

http://cnblogs.com/mjios

码拉松

小码哥教育 SEEMYGO

Array的常见操作

```swift
var arr = [1, 2, 3, 4]
// [2, 4, 6, 8]
var arr2 = arr.map { $0 * 2 }
// [2, 4]
var arr3 = arr.filter { $0 % 2 == 0 }
// 10
var arr4 = arr.reduce(0) { $0 + $1 }
// 10
var arr5 = arr.reduce(0, +)
```

```swift
func double(_ i: Int) -> Int { i * 2 }
var arr = [1, 2, 3, 4]
// [2, 4, 6, 8]
print(arr.map(double))
```

```swift
var arr = [1, 2, 3]
// [[1], [2, 2], [3, 3, 3]]
var arr2 = arr.map { Array.init(repeating: $0, count: $0) }
// [1, 2, 2, 3, 3, 3]
var arr3 = arr.flatMap { Array.init(repeating: $0, count: $0) }
```

```swift
var arr = ["123", "test", "jack", "-30"]
// [Optional(123), nil, nil, Optional(-30)]
var arr2 = arr.map { Int($0) }
// [123, -30]
var arr3 = arr.compactMap { Int($0) }
```

```swift
// 使用reduce实现map、filter的功能
var arr = [1, 2, 3, 4]
// [2, 4, 6, 8]
print(arr.map { $0 * 2 })
print(arr.reduce([]) { $0 + [$1 * 2] })

// [2, 4]
print(arr.filter { $0 % 2 == 0 })
print(arr.reduce([]) { $1 % 2 == 0 ? $0 + [$1] : $0 })
```

小码哥教育 SEEMYGO

lazy的优化

```
let arr = [1, 2, 3]
let result = arr.lazy.map {
    (i: Int) -> Int in
    print("mapping \(i)")
    return i * 2
}
print("begin-----")
print("mapped", result[0])
print("mapped", result[1])
print("mapped", result[2])
print("end----")
```

```
begin-----
mapping 1
mapped 2
mapping 2
mapped 4
mapping 3
mapped 6
end----
```

小码哥教育 SEEMYGO

Optional的map和flatMap

```swift
var num1: Int? = 10
// Optional(20)
var num2 = num1.map { $0 * 2 }

var num3: Int? = nil
// nil
var num4 = num3.map { $0 * 2 }
```

```swift
var fmt = DateFormatter()
fmt.dateFormat = "yyyy-MM-dd"
var str: String? = "2011-09-10"
// old
var date1 = str != nil ? fmt.date(from: str!) : nil
// new
var date2 = str.flatMap(fmt.date)
```

```swift
var num1: Int? = 10
// Optional(Optional(20))
var num2 = num1.map { Optional.some($0 * 2) }
// Optional(20)
var num3 = num1.flatMap { Optional.some($0 * 2) }
```

```swift
var score: Int? = 98
// old
var str1 = score != nil ? "socre is \(score!)" : "No score"
// new
var str2 = score.map { "score is \($0)" } ?? "No score"
```

```swift
var num1: Int? = 10
var num2 = (num1 != nil) ? (num1! + 10) : nil
var num3 = num1.map { $0 + 10 }
// num2、num3是等价的
```

小码哥教育 SEEMYGO

Optional的map和flatMap

```
struct Person {
    var name: String
    var age: Int
}
var items = [
    Person(name: "jack", age: 20),
    Person(name: "rose", age: 21),
    Person(name: "kate", age: 22)
]
// old
func getPerson1(_ name: String) -> Person? {
    let index = items.firstIndex { $0.name == name }
    return index != nil ? items[index!] : nil
}
// new
func getPerson2(_ name: String) -> Person? {
    return items.firstIndex { $0.name == name }.map { items[$0] }
}
```

Optional的map和flatMap

```
struct Person {
    var name: String
    var age: Int
    init?(_ json: [String : Any]) {
        guard let name = json["name"] as? String,
              let age = json["age"] as? Int else {
            return nil
        }
        self.name = name
        self.age = age
    }
}
var json: Dictionary? = ["name" : "Jack", "age" : 10]
// old
var p1 = json != nil ? Person(json!) : nil
// new
var p2 = json.flatMap(Person.init)
```

函数式编程（Funtional Programming）

- 函数式编程（Funtional Programming，简称FP）是一种编程范式，也就是如何编写程序的方法论
  - 主要思想：把计算过程尽量分解成一系列可复用函数的调用
  - 主要特征：函数是“第一等公民”
    - ✓ 函数与其他数据类型一样的地位，可以赋值给其他变量，也可以作为函数参数、函数返回值

- 函数式编程最早出现在LISP语言，绝大部分的现代编程语言也对函数式编程做了不同程度的支持，比如
  - Haskell、JavaScript、Python、Swift、Kotlin、Scala等

- 函数式编程中几个常用的概念
  - Higher-Order Function、Function Currying
  - Functor、Applicative Functor、Monad

- 参考资料
  - http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html
  - http://www.mokacoding.com/blog/functor-applicative-monads-in-pictures

小码哥教育 SEEMYGO

FP实践 – 传统写法

```
// 假设要实现以下功能：[(num + 3) * 5 - 1] % 10 / 2
var num = 1
```

```
func add(_ v1: Int, _ v2: Int) -> Int { v1 + v2 }
func sub(_ v1: Int, _ v2: Int) -> Int { v1 - v2 }
func multiple(_ v1: Int, _ v2: Int) -> Int { v1 * v2 }
func divide(_ v1: Int, _ v2: Int) -> Int { v1 / v2 }
func mod(_ v1: Int, _ v2: Int) -> Int { v1 % v2 }
```

```
divide(mod(sub(multiple(add(num, 3), 5), 1), 10), 2)
```

小码哥教育 SEEMYGO

FP实践 – 函数式写法

```
func add(_ v: Int) -> (Int) -> Int { { $0 + v } }
func sub(_ v: Int) -> (Int) -> Int { { $0 - v } }
func multiple(_ v: Int) -> (Int) -> Int { { $0 * v } }
func divide(_ v: Int) -> (Int) -> Int { { $0 / v } }
func mod(_ v: Int) -> (Int) -> Int { { $0 % v } }
```

```
infix operator >>> : AdditionPrecedence
func >>><A, B, C>(_ f1: @escaping (A) -> B,
                  _ f2: @escaping (B) -> C) -> (A) -> C { { f2(f1($0)) } }
```

```
var fn = add(3) >>> multiple(5) >>> sub(1) >>> mod(10) >>> divide(2)
fn(num)
```

高阶函数（Higher-Order Function）

- 高阶函数是至少满足下列一个条件的函数:
  - 接受一个或多个函数作为输入（map、filter、reduce等）
  - 返回一个函数

- FP中到处都是高阶函数

```
func add(_ v: Int) -> (Int) -> Int { { $0 + v } }
```

柯里化（Currying）

- 什么是柯里化？
  - 将一个接受多参数的函数变换为一系列只接受单个参数的函数

```
func add(_ v1: Int, _ v2: Int) -> Int { v1 + v2 }
add(10, 20)
```

柯里化

```
func add(_ v: Int) -> (Int) -> Int { { $0 + v } }
add(10)(20)
```

- Array、Optional的map方法接收的参数就是一个柯里化函数

柯里化（Currying）

```
func add1(_ v1: Int, _ v2: Int) -> Int { v1 + v2 }
func add2(_ v1: Int, _ v2: Int, _ v3: Int) -> Int { v1 + v2 + v3 }
```

```
func currying<A, B, C>(_ fn: @escaping (A, B) -> C)
    -> (B) -> (A) -> C {
    { b in { a in fn(a, b) } }
}

func currying<A, B, C, D>(_ fn: @escaping (A, B, C) -> D)
    -> (C) -> (B) -> (A) -> D {
    { c in { b in { a in fn(a, b, c) } } }
}
```

```
let curriedAdd1 = currying(add1)
print(curriedAdd1(10)(20))
let curriedAdd2 = currying(add2)
print(curriedAdd2(10)(20)(30))
```

小码哥教育 SEEMYGO

柯里化（Currying）

```
func add(_ v1: Int, _ v2: Int) -> Int { v1 + v2 }
func sub(_ v1: Int, _ v2: Int) -> Int { v1 - v2 }
func multiple(_ v1: Int, _ v2: Int) -> Int { v1 * v2 }
func divide(_ v1: Int, _ v2: Int) -> Int { v1 / v2 }
func mod(_ v1: Int, _ v2: Int) -> Int { v1 % v2 }
```

```
prefix func ~<A, B, C>(_ fn: @escaping (A, B) -> C)
    -> (B) -> (A) -> C { { b in { a in fn(a, b) } } }
```

```
infix operator >>> : AdditionPrecedence
func >>><A, B, C>(_ f1: @escaping (A) -> B,
                  _ f2: @escaping (B) -> C) -> (A) -> C { { f2(f1($0)) } }
```

```
var num = 1
var fn = (~add)(3) >>> (~multiple)(5) >>> (~sub)(1) >>> (~mod)(10) >>> (~divide)(2)
fn(num)
```

函子（Functor）

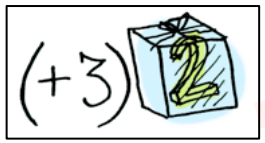
- 像Array、Optional这样支持map运算的类型，称为函子（Functor）

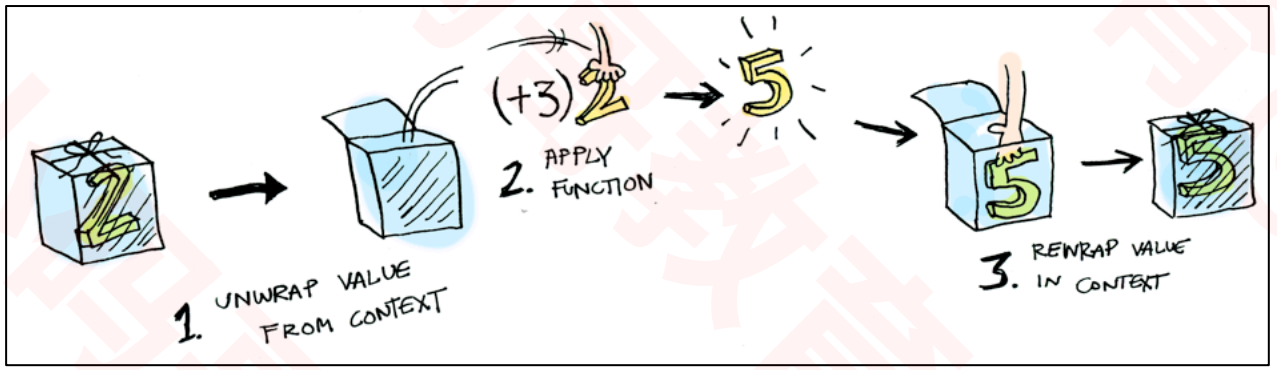
```
// Array<Element>
public func map<T>(_ transform: (Element) -> T) -> Array<T>
```

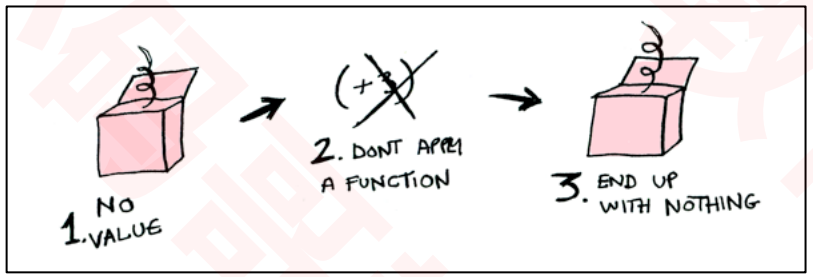
```
// Optional<Wrapped>
public func map<U>(_ transform: (Wrapped) -> U) -> Optional<U>
```

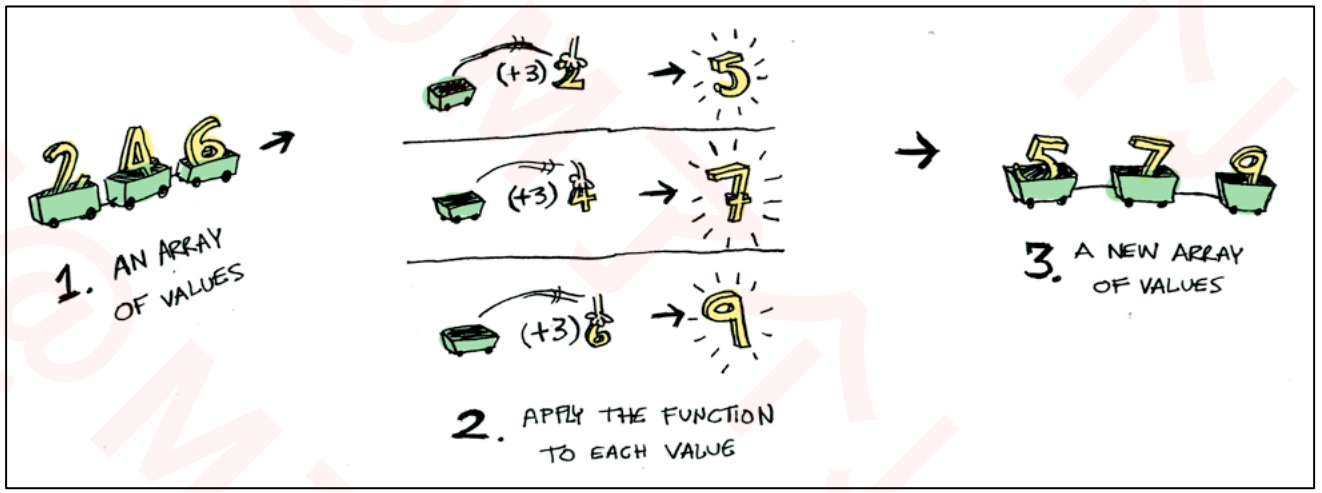
小码哥教育 SEEMYGO

函子（Functor）









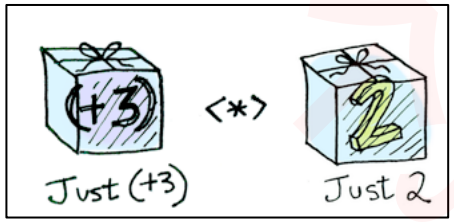
适用函子（Applicative Functor）

- 对任意一个函子 F，如果能支持以下运算，该函子就是一个适用函子

```
func pure<A>(_ value: A) -> F<A>
func <*><A, B>(fn: F<(A) -> B>, value: F<A>) -> F<B>
```

- Optional可以成为适用函子

```
func pure<A>(_ value: A) -> A? { value }
infix operator <*> : AdditionPrecedence
func <*><A, B>(fn: ((A) -> B)?, value: A?) -> B? {
    guard let f = fn, let v = value else { return nil }
    return f(v)
}
```




```
var value: Int? = 10
var fn: ((Int) -> Int)? = { $0 * 2}
// Optional(20)
print(fn <*> value as Any)
```

适用函子（Applicative Functor）

- Array可以成为适用函子

```
func pure<A>(_ value: A) -> [A] { [value] }
func <*><A, B>(fn: [(A) -> B], value: [A]) -> [B] {
    var arr: [B] = []
    if fn.count == value.count {
        for i in fn.startIndex..<fn.endIndex {
            arr.append(fn[i](value[i]))
        }
    }
    return arr
}
```

```
// [10]
print(pure(10))

var arr = [{ $0 * 2}, { $0 + 10 }, { $0 - 5 }] <*> [1, 2, 3]
// [2, 12, -2]
print(arr)
```

单子（Monad）

- 对任意一个类型 F，如果能支持以下运算，那么就可以称为是一个单子（Monad）

```
func pure<A>(_ value: A) -> F<A>
func flatMap<A, B>(_ value: F<A>, _ fn: (A) -> F<B>) -> F<B>
```

- 很显然，Array、Optional都是单子