Урок 8

Функциональный Swift

Использование подходов функционального программирования в Swift. Функторы.

Парадигмы программирования

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания программ, то есть это определенный подход к разработке. Чтобы использовать ту или иную идею, концепцию, парадигму (сейчас их множество), необходимо время, документация, практики использования и главное — понимание у разработчиков, к каким наборам задач можно применить ту или иную парадигму.

Если мы заглянем в историю, увидим, что процедурное программирование отлично справлялось со своими задачами. Оно было очень быстрым.

Процедурное программирование

Процедурное программирование основано на разбивке общего алгоритма на несколько независимых модулей — процедур. Комбинируя процедуры в основном модуле программы, можно получить алгоритмы любой сложности. Задачи решаются шаг за шагом. Выполнение программы сводится к последовательному выполнению процедур с целью преобразовать исходное состояние данных в результирующее. По парадигме процедурного программирования программа последовательно изменяет содержимое памяти.

Затем на смену идеологии процедурного программирования пришла парадигма ООП (объектно-ориентированное программирование). Первым объектно-ориентированным языком, реализовавшим основные идеи ООП, был Симула (Simula), появившийся в 1967 году. Тогда никто не понимал, зачем этот язык нужен. Обращали внимание лишь на потери в производительности (нахождение верной реализации метода, правильного наследника). Но программы становились все сложнее, команды разработчиков увеличивались, и оказалось, что при этих условиях не хватало объектно-ориентированного подхода. В итоге парадигма ООП стала популярной и окончательно закрепилась с появлением языка C++, представленного в 1983 году.

ООП

Объектно-ориентированное программирование (ООП) — парадигма, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования. Парадигма ООП предлагает использовать объектное мышление, чтобы найти нужные абстракции объектов для решения задач. От того, насколько удачно выбраны абстракции, зависит наше видение программы в целом. Это позволяет быстро разобраться, что происходит в системе. Парадигма ООП подразумевает следование основным принципам — инкапсуляции, наследованию и полиморфизму. Они нужны для удобного расширения программы без дублирования кода. По парадигме ООП состояние данных хранится в объектах.

ПОП

Протокольно-ориентированное программирование (ПОП) — парадигма программирования, развивающая ООП. Гласит, что нужно не наследоваться, а использовать протоколы. Принципы парадигмы ПОП стали в полной мере доступны в Swift после обновления до версии Swift 2 и появления в ней возможности расширения протокола.

Рассмотрим пример преобразования кода к парадигме ПОП. Предположим, что есть некоторый класс **SomeImageView** с методом **shake()** и класс контроллера **SomeViewController,** содержащий переменную этого класса и вызывающий метод **shake()**.

```
class SomeImageView: UIImageView {
    func shake() {
        let animation = CABasicAnimation(keyPath: "position")
        animation.duration = 0.07
        animation.repeatCount = 3
        animation.autoreverses = true
        animation.fromValue = NSValue(cgPoint: CGPoint(x: self.center.x - 10, y:
self.center.y))
        animation.toValue = NSValue(cgPoint: CGPoint(x: self.center.x + 10, y:
self.center.y))
        self.layer.add(animation, forKey: "position")
    }
}

class SomeController: UIViewController {

    @IBOutlet weak var imageView: SomeImageView!

    @IBAction func tap(sender: AnyObject) {
        imageView.shake()
    }
}
```

Далее предположим, что дрожать должна еще одна кнопка, и появляется еще один класс с аналогичной реализацией метода **shake()**.

```
class SomeImageView: UIImageView {
    func shake() {
        let animation = CABasicAnimation(keyPath: "position")
        animation.duration = 0.07
        animation.repeatCount = 3
        animation.autoreverses = true
        animation.fromValue = NSValue(cgPoint: CGPoint(x: self.center.x - 10, y:
self.center.y))
        animation.toValue = NSValue(cgPoint: CGPoint(x: self.center.x + 10, y:
self.center.y))
        self.layer.add(animation, forKey: "position")
    }
}

class SomeButton: UIButton {
    func shake() {
        let animation = CABasicAnimation(keyPath: "position")
        animation.duration = 0.07
        animation.repeatCount = 3
        animation.autoreverses = true
        animation.fromValue = NSValue(cgPoint: CGPoint(x: self.center.x - 10, y:
self.center.y))
        animation.toValue = NSValue(cgPoint: CGPoint(x: self.center.x + 10, y:
self.center.y))
        self.layer.add(animation, forKey: "position")
    }
}

class SomeController: UIViewController {

    @IBOutlet weak var imageView: SomeImageView!
    @IBOutlet weak var button: SomeButton!
```

```
    @IBAction func tap(sender: AnyObject) {
        imageView.shake()
        button.shake()
    }
}
```

В результате получим дублирование. Будучи хорошими программистами, попытаемся избавиться от него и перенесем реализацию метода **shake()** в расширение **UIView**.

```
extension UIView {
    func shake() {
        let animation = CABasicAnimation(keyPath: "position")
        animation.duration = 0.07
        animation.repeatCount = 3
        animation.autoreverses = true
        animation.fromValue = NSValue(cgPoint: CGPoint(x: self.center.x - 10, y:
self.center.y))
        animation.toValue = NSValue(cgPoint: CGPoint(x: self.center.x + 10, y:
self.center.y))
        self.layer.add(animation, forKey: "position")
    }
}

class SomeImageView: UIImageView {
}

class SomeButton: UIButton {
}

class SomeController: UIViewController {

    @IBOutlet weak var imageView: SomeImageView!
    @IBOutlet weak var button: SomeButton!

    @IBAction func tap(sender: AnyObject) {
        imageView.shake()
        button.shake()
    }
}
```

Но так потеряем наглядность кода. На помощь придет парадигма ПОП. В соответствии с ней будем использовать протокол вместо расширения.

```
protocol Shakeable { }

extension Shakeable where Self: UIView {
    func shake() {
        let animation = CABasicAnimation(keyPath: "position")
        animation.duration = 0.07
        animation.repeatCount = 3
        animation.autoreverses = true
        animation.fromValue = NSValue(cgPoint: CGPoint(x: self.center.x - 10, y:
self.center.y))
        animation.toValue = NSValue(cgPoint: CGPoint(x: self.center.x + 10, y:
self.center.y))
        self.layer.add(animation, forKey: "position")
```

```
    }
}

class SomeImageView: UIImageView, Shakeable {
}

class SomeButton: UIButton, Shakeable {
}

class SomeController: UIViewController {

    @IBOutlet weak var imageView: SomeImageView!
    @IBOutlet weak var button: SomeButton!

    @IBAction func tap(sender: AnyObject) {
        imageView.shake()
        button.shake()
    }
}
```

В результате подхода ПОП мы получили более читабельную версию кода.

По примеру использования парадигмы ПОП можно в наших программах использовать и другие подходы, в частности — парадигму функционального программирования.

Функциональное программирование

Функциональное программирование основано на использовании функций. Функция — это базовый элемент функционального программирования. Функции используются почти для всего, даже для простейших вычислений и замены переменных. Их нельзя изменять, и в каждую переменную можно записать только один раз. В функциональном программировании можно хранить состояние, используя не переменные, а функции. Состояние хранится в параметрах функции, в стеке. Если нужно сохранить состояние, чтобы потом изменить его, нужно написать рекурсивную функцию.

Основные преимущества функционального программирования:

- **Никаких побочных эффектов**. Функция не может поменять значение вне своей области видимости и повлиять на другие функции (как это может случиться с полями класса или глобальными переменными);
- **Модульное тестирование**. Единственный результат выполнения функции — это возвращаемое значение, а повлиять на возвращаемое значение могут только аргументы. Можно протестировать каждую функцию в программе, используя разные вариации аргументов;
- **Отладка**. Мы всегда можем воспроизвести ошибку, потому что баг в функции не зависит от постороннего кода, который выполнялся ранее;
- **Многопоточность**. Код сразу готов к распараллеливанию без изменений. В функциональном программировании не нужно думать ни о **deadlock**, ни о состоянии гонок, так как данные не меняются дважды в любом из потоков;
- **Функции высшего порядка**. Функции высшего порядка принимают другие функции в качестве аргументов.

- **Каррирование**. Интерфейс функции — это ее аргументы. Каррирование используется для уменьшения количества аргументов функции.

Посмотрим на применение подходов функционального программирования в Swift.

Функциональное программирование в Swift

Чистые функции

Основное правило функционального программирования: все функции — чистые.

Функция является чистой, если она удовлетворяет следующим условиям:

- Функция, вызываемая от одних и тех же аргументов, всегда возвращает одинаковое значение;
- Во время выполнения функции не возникают **side effects** (побочные эффекты). Это любые действия, изменяющие среду выполнения. К побочным эффектам относят модифицирование глобальных переменных, изменение объектов, изменение контекста файлов.

Может показаться, что это серьезное ограничение. Но если мы уверены, что вызов функции не изменит ничего «снаружи», можно использовать эту функцию где угодно. Это правило как раз и ведет к многопоточности.

Пример чистой функции:

```
func add(x: Int, y: Int) -> Int {
    return x + y
}
```

Пример нечистой функции:

```
func add(x: Int, y: Int) -> Int {
    let result = x + y

    writeToFile(result) // Побочный эффект — запись результата в файл!

    return result
}
```

Функции высшего порядка

Функции высшего порядка принимают другую функцию как аргумент или возвращают функцию. В Swift уже реализованы базовые функции из функционального программирования, такие как **map**, **filter** и **reduce** для массива.

Посмотрим пример их использования:

```
let sum = [1, 2, 3, 4].map{ $0 * $0 }.filter{ $0 % 2 == 0 }.reduce(0){ $0 + $1 }
print("sum = \(sum)")
```

```
// sum = 20
```

Функция **map** перегоняет коллекцию объектов одного типа в коллекцию объектов другого. В функцию **filter** мы передаем замыкание, которое определяет, включать ли значение в результирующую выборку. Функция **reduce** сжимает массив до одного значения.

Рассмотрим несколько примеров рефакторинга кода с использованием функций высшего порядка.

Пример первый. До:

```
class FormSelectItem {
    var value: String?

    ...
}

func createChoiceItems(catalog: [FormSelectItem]?) -> [String] {
    var data: [String] = []
    if let items = catalog {
        items.forEach { item in
            if let itemValue = item.value {
                data.append(itemValue)
            }
        }
    }
    return data
}
```

Код в приведенном выше примере правильный. Но посмотрим на тот же метод, только с применением **compactMap()**.

После:

```
class FormSelectItem {
    var value: String?

  ...
}

func createChoiceItems(catalog: [FormSelectItem]?) -> [String] {
    return catalog?.compactMap { $0.value } ?? []
}
```

Код уменьшился в разы и стал более читабельным.

Пример второй. До:

```
class Item {
    var value: Int?
    var selectItems: [Item] = []

    init(value: Int?) {
        self.value = value
    }
}
```

```
var formFieldItem: Item? = Item(value: 1)

var selectedIndex: Int? = 0
if  let selectedItem = formFieldItem?.value,
    let selectItems = formFieldItem?.selectItems {
    for (index, selectItem) in selectItems.enumerated() {
        if let selectItemValue = selectItem.value {
            if selectedItem == selectItemValue { selectedIndex = index }
        }
    }
}
```

После:

```
class Item {
    var value: Int?
    var selectItems: [Item] = []

    init(value: Int?) {
        self.value = value
    }
}

var formFieldItem: Item? = Item(value: 1)

let selectedIndex = formFieldItem?.selectItems
    .enumerated()
    .first(where: { $0.element.value == formFieldItem?.value})
    .map { $0.offset }
```

Как и в первом примере, наблюдаем уменьшение кода без потери в его понятности.

Рекурсия

Рекурсия — это повторение операции. Это достаточно распространенное явление, которое встречается и в повседневной жизни. Например, если навести web-камеру на экран монитора компьютера, она будет записывать изображение экрана и выводить на него же. Получится что-то вроде замкнутого цикла. В итоге мы будем наблюдать нечто, похожее на тоннель.

В функциональном программировании рекурсия — это функция, которая вызывает саму себя непосредственно в своем теле или через другую функцию.

В уроке 7 мы уже приводили пример рекурсивной функции при расчете последовательности чисел Фибоначчи с заданным шагом:

```
// Рекурсивная функция
func fibonacciRecursive(number1: Int, number2: Int, steps: Int, result: inout
[Int]) {
    if steps > 0 {
        let nextNumber = number1 + number2
        result.append(nextNumber)
        fibonacciRecursive(number1: number2, number2: nextNumber, steps:
steps-1, result: &result)
    }
}
```

```
func fibonacci(steps: Int) -> [Int] {
    let defaultFirstNumber = 0
    let defaultSecondNumber = 1
    var result = [defaultFirstNumber, defaultSecondNumber]

    fibonacciRecursive(number1: defaultFirstNumber, number2:
defaultSecondNumber, steps: steps, result: &result)

    return result
}


print(fibonacci(steps: 7))

// [0, 1, 1, 2, 3, 5, 8, 13, 21]
```

Каррирование

Каррирование — это способ создания функций, позволяющий частичное применение их аргументов. Можно передать все аргументы, ожидаемые функцией, и получить результат, или передать часть этих аргументов и получить обратно функцию, ожидающую остальные аргументы.

Рассмотрим пример функции, выводящей в **output** приветствие:

```
func greetMen(greeting: String, separator: String, name: String) {
    print("\(greeting)\(separator)\(name)")
}

greetMen(greeting: "Hello", separator: ",", name: "Alex")

// Hello,Alex
```

Рассмотрим вариант этой функции с использованием каррирования:

```
func greetMen(_ greeting: String) -> (String) -> ((String) -> ()) {
    return { separator in
        return { name in
            print("\(greeting)\(separator)\(name)")
        }
    }
}

greetMen("Hello")(",")("Alex")

// Hello,Alex
```

Каррирование — полезный способ в функциональном программировании, так как позволяет конфигурировать функции, которые быстро используются и поняты при чтении кода.

Функторы

Дадим строгое определение. Объект F является функтором, если он выполняет следующее условие:

```
F[A].map(A → B) = F[B]
```

То есть функтор — это контейнер, к которому применима функция **map**.

Распространенные типы функторов: Result, Either, Promise

Рассмотрим работу с функторами на примере функции деления двух чисел, в которой есть «подводный камень» — деление на ноль, что приводит к аварийному завершению приложения. Учтем этот факт при создании функции и применим использование кортежей для обработки такой ситуации.

```
extension NSError {
    convenience init(localizedDescription: String) {
        self.init(domain: "", code: 1, userInfo: [NSLocalizedDescriptionKey:
localizedDescription])
    }
}

func divide (x:Float, y:Float) -> (result: Float?, error: NSError?){
    if y == 0 {
        return (nil, NSError(localizedDescription: "Деление на ноль"))
    } else {
        return (x/y, nil)
    }
}

var value = divide(x: 2, y: 1)
print("divide(x: 2, y: 1) result: \(value.result ?? 0)")
print("divide(x: 2, y: 1) error: \(value.error?.localizedDescription ?? "")")

value = divide(x: 3, y: 0)
print("divide(x: 3, y: 0) result: \(value.result ?? 0)")
print("divide(x: 3, y: 0) error: \(value.error?.localizedDescription ?? "")")

// divide(x: 2, y: 1) result: 2.0
// divide(x: 2, y: 1) error:
// divide(x: 3, y: 0) result: 0.0
// divide(x: 3, y: 0) error: Деление на ноль
```

Swift позволяет использовать в этом случае не кортежи, а **enum**:

```
enum Result<T> {
    case Success(T)
    case Failure(String)

    var description : String {
        get {
            switch self {
            case .Success(let value):
                return  "\(value)"
            case .Failure(let message):
                return "\(message)"
            }
        }
    }
}
```

```
func divide (x:Float, y:Float) -> Result<Float> {
    if y == 0 {
        return .Failure("Деление на ноль")
    } else {
        return .Success(x/y)
    }
}

var result = divide(x: 2, y: 1)
print("divide(x: 2, y: 1) result: \(result.description)")

result = divide(x: 3, y: 0)
print("divide(x: 3, y: 0) result: \(result.description)")

// divide(x: 2, y: 1) result: 2.0
// divide(x: 3, y: 0) result: Деление на ноль
```

Код получился гораздо нагляднее. Switch позволяет связать значения с именами и обращаться к ним в коде. Результат же можно обрабатывать в зависимости от возникновения ошибки.

Теперь предположим, что нам потребуется сделать вычисления с результатом деления, а именно — выполнить функции **square()** и **sum10()**.

```
enum Result<T> {
    case Success(T)
    case Failure(String)

    var description : String {
        get {
            switch self {
            case .Success(let value):
                return  "\(value)"
            case .Failure(let message):
                return "\(message)"
            }
        }
    }
}

func divide(x:Float, y:Float) -> Result<Float> {
    if y == 0 {
        return .Failure("Деление на ноль")
    } else {
        return .Success(x/y)
    }
}

func square(divideResult:Result<Float>) -> Result<Float> {
    switch divideResult {
    case .Success(let value):
        return Result.Success(value * value)

    case .Failure(let errString):
        return Result.Failure(errString)
    }
}

func sum10(divideResult:Result<Float>) -> Result<Float> {
```

```
    switch divideResult {
    case .Success(let value):
        return Result.Success(value + 10)

    case .Failure(let errString):
        return Result.Failure(errString)
    }
}

var result = square(divideResult: divide(x: 2, y: 1))
print("result: \(result.description)")

result = sum10(divideResult: divide(x: 3, y: 0))
print("result: \(result.description)")

// result: 4.0
// result: Деление на ноль
```

Наблюдаем дублирование кода в функциях **square()** и **sum10()**, связанных с проверкой **divideResult**. Нужна абстракция. Перечисления могут иметь методы. Реализуем метод **map()**.

```
enum Result<T> {
    case Success(T)
    case Failure(String)

    func map<P>(f: (T) -> P) -> Result<P> {
        switch self {
        case .Success(let value):
            return .Success(f(value))
        case .Failure(let err):
            return .Failure(err)
        }
    }

    var description : String {
        get {
            switch self {
            case .Success(let value):
                return  "\(value)"
            case .Failure(let message):
                return "\(message)"
            }
        }
    }
}

func square(x:Float) -> Float {
    return x * x
}

func sum10(x:Float) -> Float {
    return x + 10
}

func divide (x:Float, y:Float) -> Result<Float> {
    if y == 0 {
        return .Failure("Деление на ноль")
```

```
    } else {
        return .Success(x/y)
    }
}

var result = divide(x: 2, y: 1).map(f: square).map(f: sum10)
print("result: \(result.description)")

result = divide(x: 2, y: 0).map(f: square).map(f: sum10)
print("result: \(result.description)")

// result: 14.0
// result: Деление на ноль
```

Получили функтор с использованием функционального программирования.

На самом деле мы использовали функтор типа **Result**. Его общий вид:

```
enum Result<A> {
  case Error(String)
  case Value(A)
}
```

Тип **Result** является частным случаем типа функтора **Either**. Его общий вид:

```
enum Either<A, B> {
  case Left(A)
  case Right(B)
}
```

Тип **Promise** обрабатывает тип **T** и возвращает новую упаковку **Promise<U>** нового типа. Его общий вид выглядит так:

```
class Promise<T> {
    func map<U>(f: T -> U) -> Promise<U>
    func flatMap<U>(f: T -> Promise<U>) -> Promise<U>
}
```

Аппликативные функторы

Объект **F** является аппликативным функтором, если он выполняет следующее условие:

```
F[A].apply( F[A → B] ) = F[B]
```

Рассмотрим пример аппликативного функтора на Swift:

```
class Box<T> {
    var content: T
    init(content: T) {
        self.content = content
    }
```

```
    func apply<U>(_ relation: Box<(T) -> U>) -> Box<U> {
        return Box<U>(content: relation.content(self.content))
    }
}

let box: Box<(Int)->Int> = Box(content: {$0 + 5} )
let content = Box(content: 1).apply(box).apply(box).content
print("content: \(content)")

// content: 11
```

Optional, Array — функторы из Swift

Вооружившись знаниями о функторах, опять замечаем следы функционального программирования в Swift.

Исходя из документации по опциональным типам, если мы применяем представленную ниже запись — на самом деле используем короткую форму записи опционала:

```
let value: Int? = Int("1")
```

Полная же форма записи выглядит следующим образом:

```
let value: Optional<Int> = Int("1")
```

Если мы «провалимся» по типу **Optinal** в XCode, увидим, что **Optinal** — это enum-тип и функтор:

```
public enum Optional<Wrapped> : ExpressibleByNilLiteral {

    /// The absence of a value.
    ///
    /// In code, the absence of a value is typically written using the `nil`
    /// literal rather than the explicit `.none` enumeration case.
    case none

    /// The presence of a value, stored as `Wrapped`.
    case some(Wrapped)
```

Тип **Array** полностью соответствует определению функтора, данному выше.

Резюмируем: не нужно срочно переходить с любимого ООП на функциональное программирование. Все же время повсеместного применения функционального программирования прошло. Но у нас есть большой список задач (обработка данных, фильтрация списков), которые хорошо решаются при помощи функционального программирования.

Практическое задание

1. Определить, какие участки вашего кода лучше переписать с использованием функционального программирования.

2. Провести рефакторинг найденных участков.
3. Реализовать экран списка товаров.

Дополнительные материалы

1. Функциональный язык программирования Haskell.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Error Handling in Swift: Might and Magic.