SWIFT
函数式编程
实践

@HANKBAO

WHAT

WHAT

Functional programming is a programming *paradigm*
1. treats computation as the evaluation of mathematical functions
2. avoids changing-state and mutable data
— Wikipedia

PARADIGM

WHY

高阶函数

FILTER/MAP/REDUCE...

```swift
let nums = [1, 2, 3, 4, 5, 6, 7]

var strs = [String]()
for num in nums {
    strs.append(String(num))
}
```

```swift
let nums = [1, 2, 3, 4, 5, 6, 7]
let strs = nums.map(String.init)
```

CURRY

```
func x(a: A, b: B, c: C) -> E

func x(a: A) -> (b: B) -> (c: C) -> E

func curry<A, B, C, E>(f: (A, B, C) -> E) -> A -> B -> C -> E {
    return { a in { b in { c in f(a, b, c) }}}
}
```

```swift
class User {
    func login(password: String)
}

let passwd = "@Swift"
let usr = User()

usr.login(passwd)
      ||
User.login(usr)(passwd)
```

```swift
class User {
    func name() -> String
}

let collation: UILocalizedIndexedCollation = ...

let sorted = collation.sortedArrayFromArray(users,
              collationStringSelector: "name")
```

```swift
class Wrapper<T>: NSObject {
    let payload: T
    let localization: (T) -> () -> String

    @objc func localizable() -> NSString {
        return localization(payload)()
    }

    static var selector: Selector {
        return "localizable"
    }
}
```

```swift
let wrappers = users.map {
    Wrapper(payload: $0, localization: User.name)
}

let sorted = collation.sortedArrayFromArray(wrappers,
      collationStringSelector: Wrapper<User>.selector)
```

递归

```swift
indirect enum Tree<Element> {
    case Empty
    case Node(Element, Tree<Element>, Tree<Element>)
}
```

```swift
func reversedTree<T>(tree: Tree<T>) -> Tree<T> {
    switch tree {
    case .Empty:
        return .Empty

    case let .Node(element, left, right):
        return .Node(element, reversedTree(right), reversedTree(left))
    }
}
```

OPTIONAL

OPTIONAL

```
enum Optional<T> {
   case None
   case Some(T)
}

func map<U>(f: Wrapped -> U) -> U?
func flatMap<U>(f: Wrapped -> U?) -> U?
```

OPTIONAL

```
let date: NSDate? = ...
let formatter: NSDateFormatter = ...

let dateString = date.map(formatter.stringFromDate)
```

ARRAY (SEQUENCETYPE)

```
func map<T>(t: Element -> T) -> [T]
func flatMap<S>(t: Element -> S) -> [S.Element]
```

MONAD

```swift
enum Result<Value> {
    case Failure(ErrorType)
    case Success(Value)
}
```

```swift
(value: T?, error: ErrorType?) -> Void

if let error = error {
    // handle error
} else if let value = value {
    // handle value
} else {
    // all nil?
}

// all non-nil?!
```

```swift
(result: Result<T>) -> Void

switch result {
case let .Error(error):
    // handle error
case let .Success(value):
    // handle value
}
```

```swift
enum Result<Value> {
    func map<T>(...) -> Result<T> {
       ...
    }

    func flatMap<T>(...) -> Result<T> {
       ...
   }
}
```

```swift
func flatMap<T>(@noescape transform: Value throws -> Result<T>) rethrows -> Result<T> {
    switch self {
    case let .Failure(error):
        return .Failure(error)

    case let .Success(value):
        return try transform(value)
    }
}

func map<T>(@noescape transform: Value throws -> T) rethrows -> Result<T> {
    return try flatMap { .Success(try transform($0)) }
}
```

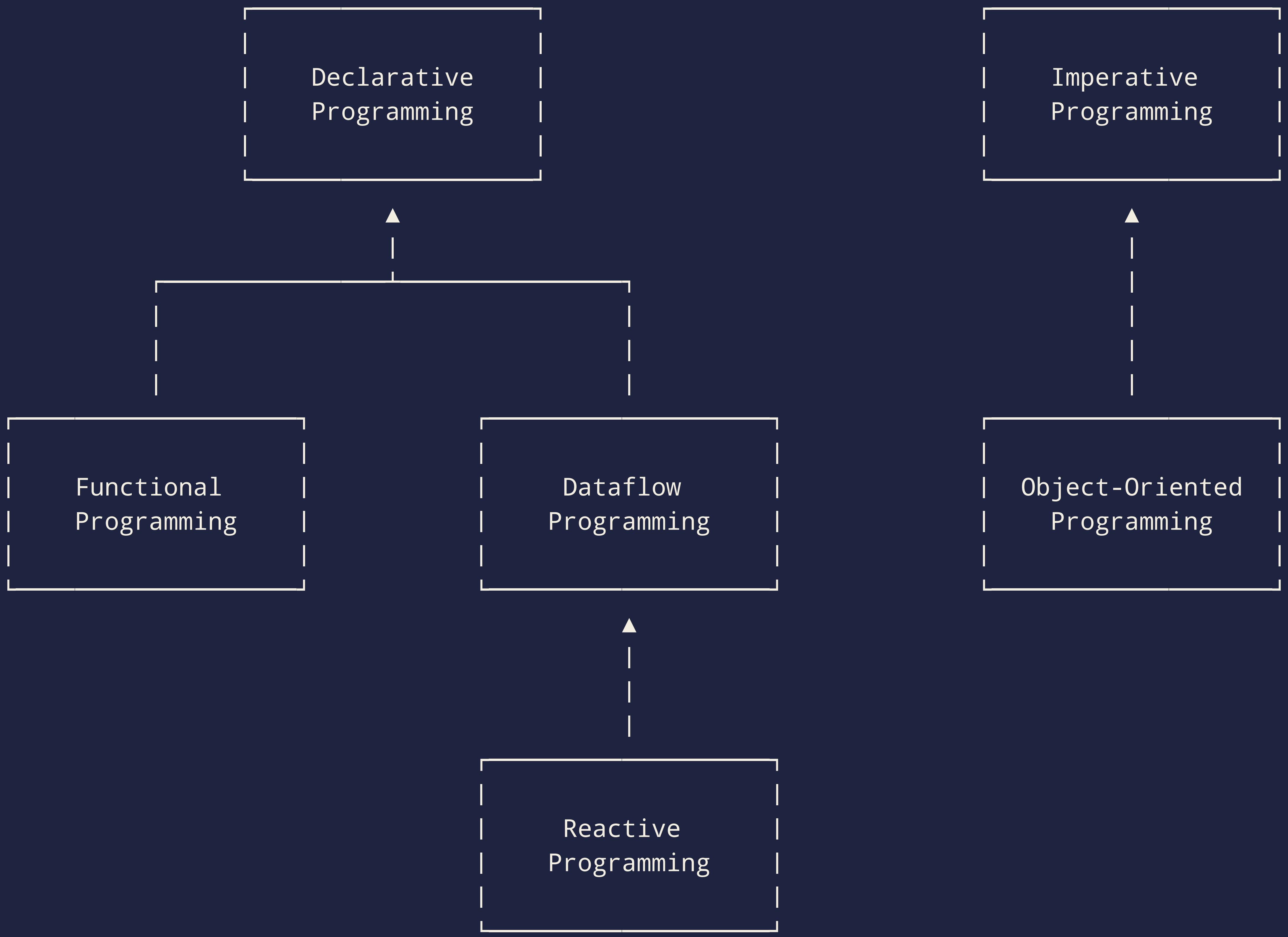
```swift
func toImage(data: NSData) -> Result<UIImage>
func addAlpha(image: UIImage) -> Result<UIImage>
func roundCorner(image: UIImage) -> Result<UIImage>
func applyBlur(image: UIImage) -> Result<UIImage>

toImage(data)
  .flatMap(addAlpha)
  .flatMap(roundCorner)
  .flatMap(applyBlur)
```

PROMISE

```
class Promise<T> {
    func then<U>(body: T -> U) -> Promise<U>
    func then<U>(body: T -> Promise<U>) -> Promise<U>
}
```

MONAD & OBSERVABLE


Declarative Programming
Imperative Programming
Functional Programming
Dataflow Programming
Object-Oriented Programming
Reactive Programming

思维

参考资料

- Wikipedia
- Haskell Wiki
- Functional Programming in Swift
- objc.io

THANKS
Q & A