Swiftによる
関数型プログラミング
超入門

2015年11月14日

藤本尚邦 / Cocoa勉強会(関東) #75

自己紹介

- 藤本尚邦 (@fhisa)
- https://github.com/fhisa
- フリーランスプログラマー
- RubyCocoaフレームワーク原作者
- Mac開発歴、薄く長く約25年
- iOS開発歴、約1年

アジェンダ

- 始めに
- 総和・総乗
- 総和・総乗の抽象化
- SequenceTypeプロトコル
- 関数型プログラミングとは
- まとめ

始めに

- 関数型プログラミング[1]ってよく聞くけど何それ？
- C言語には関数があるから関数型プログラミングができるの？
- `map` とか `reduce` って何？どうやって使うの？

このような立ち位置の人を想定して、関数型プログラミング風の考え方の一端を紹介します。

---

[1] 英語で *Fanctional Programming* といいます。短縮して *FP* と呼ばれています。

始めに

といっても私自身、本当は語れるほど*FP*を理解しているわけではありません。ただし、以前自分でもそれと気付かずに*FP*を学んだことがあります。

その経験をもとに発表の構成を考えました。

総和・総乗

総和・総乗

定義 1 :

$$f(n) = \sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n$$

$$f(n) = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \cdots \times n$$

総和・総乗

```
func sum(n: Int) -> Int {
    var accumulator = 0
    for i in 1...n { accumulator += i }
    return accumulator
}
```

```
func prod(n: Int) -> Int {
    var accumulator = 1
    for i in 1...n { accumulator *= i }
    return accumulator
}
```

ループと代入を使った手続き的なプログラム

総和・総乗

定義２:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ n + f(n-1) & \text{if } n > 0 \end{cases}$$

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n-1) & \text{if } n > 0 \end{cases}$$

総和・総乗

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

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

再帰呼出を使った*FP*の香り漂うプログラム

総和 (番外編)

ガウス(小３)による総和のプログラム:

```
func sum(n: Int) -> Int {
    return n * (n + 1) / 2
}
```

総和についてはガウスの方法が賢いやり方だが、総乗への応用は多分きかない。

総和・総乗の抽象化

総和・総乗の抽象化

２つの関数の似ている点・違う点を探してみよう:

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

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

総和・総乗の抽象化

２つの関数の違う点:

- n==0のとき、前者は0を返し後者は1を返す
- n!=0のとき、前者は足し算で後者は掛け算

総和・総乗の抽象化

元の引数nに加えて、総和・総乗で違う以下の２点:

- n==0のときの値 (初期値)
- n!=0のときの計算 (関数)

を引数にすれば、同じパターンの計算に使えるより汎用的な関数が出来上がるはずです。

総和・総乗の抽象化

```
typealias IntCombinator = (Int, Int) -> Int

func reduce(n: Int, initial: Int, combine: IntCombinator) -> Int
{
    if n == 0 {
        return initial
    } else {
        return combine(
            n,
            reduce(n - 1, initial: initial, combine: combine))
    }
}
```

総和・総乗のパターンを抽象化した関数 reduce

総和・総乗の抽象化

関数reduceを使って総和・総乗を計算してみよう:

```
// ふたつの整数の和算・乗算をする関数を定義
func add2(n: Int, m: Int) -> Int { return n + m }
func mul2(n: Int, m: Int) -> Int { return n * m }

reduce(10, initial: 0, combine: add2)
reduce(10, initial: 1, combine: mul2)
```

総和・総乗の抽象化

関数reduceを使って総和・総乗を計算してみよう(2):

```
// 無名関数(関数リテラル)を直接渡す
reduce(10, initial: 0, combine: { $0 + $1 })
reduce(10, initial: 1, combine: { $0 * $1 })

// Ruby風シンタックスシュガー
reduce(10, initial: 0) { $0 + $1 }
reduce(10, initial: 1) { $0 * $1 }
```

総和・総乗の抽象化

Swiftでは + や - も関数なので引数に渡せます:

```
// モダンな関数型プログラミング風
reduce(10, initial: 0, combine: +)
reduce(10, initial: 1, combine: *)
```

add2 や mul2 のような関数を定義する必要はありませんでした。

総和・総乗の抽象化

ここまで、関数 reduce の定義にいたる道筋を示すことで、関数型プログラミングの考え方の土台を簡単に紹介しました。

今回は、関数を引数にとる関数を題材にしましたが、関数を返す関数を使ったプログラミングなどさらに奥深い世界が存在しています。

SequenceTypeプロトコル

SequenceTypeプロトコル

```
for item in collection {
    statements
}
```

Swiftの For-In構文では、繰り返しの対象となる `collection` が `SequenceType` プロトコルに適合している必要があります。

SequenceTypeプロトコル

代表的なインスタンスメソッド:

- `filter`
- `map`
- `reduce`
- `sort`

SequenceTypeプロトコル

`filter`

引数で与えられた関数で各`item`を調べ、`true`を返したitemのみによる配列を返します。

```
(1...10).filter { $0 % 2 == 0 }  // => [2, 4, 6, 8, 10]
(1...10).filter { $0 % 2 != 0 }  // => [1, 3, 5, 7, 9]
```

SequenceTypeプロトコル

`map`

数学で写像(mapping)と呼ばれているものに相当します。各`item`に関数を適用した結果の配列を返します。

```
(1...5).map { $0 * 2 } // => [2, 4, 6, 8, 10]
(1...5).map { String($0) } // => ["1", "2", "3", "4", "5"]
(1...5).map(String.init)  // => ["1", "2", "3", "4", "5"]
```

SequenceTypeプロトコル

reduce

現実的・汎用的な本物のreduceです[23]。

```
(1...10).reduce(0, combine: +) // Swift的な総和の計算
(1...10).reduce(1, combine: *) // Swift的な総乗の計算

["aa", "bb", "cc"].reduce("") { "\($0)/\($1)" }
  // => "/aa/bb/cc"
```

---

[2] SequenceTypeではitemの型も変数なので、Int型に限定せず扱うことができます。

[3] 言語によっては inject (Ruby, Smalltalkなど) や fold (Haskellなど) と呼ばれています。

SequenceTypeプロトコル

sort

`item`を比較関数で並べ変えた配列を返します。`item`が`Comparable`プロトコルに適合している場合は、比較関数を省略できます。

```
[5,4,3,2,1].sort()  // => [1, 2, 3, 4, 5]
(1...5).sort(>)  // => [5, 4, 3, 2, 1]
```

SequenceTypeプロトコル

次のようなPersonと名簿が定義されているとします:

```
struct Person {
    let name: String
    let age: Int
    let sex: Sex
    enum Sex { case Male, Female, Others }
}

let BABYMETAL: [Person] = [
    Person(name: "中元すず香", age: 17, sex: .Female),
    Person(name: "菊地最愛", age: 16, sex: .Female),
    Person(name: "水野由結", age: 16, sex: .Female),
    Person(name: "青山英樹", age: 29, sex: .Male),
    Person(name: "BOH", age: 33, sex: .Male),
    Person(name: "大村孝佳", age: 31, sex: .Male),
    Person(name: "藤岡幹大", age: 34, sex: .Male),
]
```

SequenceTypeプロトコル

```
// BABYMETALの女子の名前をソート:
BABYMETAL
    .filter { $0.sex == .Female }
    .map { $0.name }
    .sort()

// BABYMETAL全員の平均年齢:
let age_sum = BABYMETAL
    .map { $0.age }
    .reduce(0, combine: +)
Float(age_sum) / Float(BABYMETAL.count)
```

関数型プログラミングとは

関数型プログラミングとは

Swiftの特徴[4]:

- 関数がファーストクラスオブジェクト
    - 関数を引数や返り値として扱える
- 関数と普通の変数の名前空間が一緒

[4] JavaScript はこの特徴を持つプログラミング言語の代表例

関数型プログラミングとは

「計算機プログラムの構造と解釈」(通称SICP)

1. 手続きを用いた抽象化の構築 (高階関数)

2. データを用いた抽象化の構築 (データ構造)

3. モジュール性、オブジェクト、状態 (代入初登場)

4. メタ言語抽象化 (プログラミング言語の実装)

5. レジスタマシンによる計算 (CPU・コンパイラ)

関数型プログラミングとは

代入を使わない限り、同じ引数による同じ手続きの呼び出しは２回評価しても同じ結果になるので、手続きは数学関数の計算とみなすことができる...まったく代入を使わないプログラミングは、そのため**関数型プログラミング**(*functional programming*)と呼ばれている。

— *Structure and Interpretation of Computer Programs* (真鍋宏史日本語訳版) より

まとめ

Swiftは手続き型プログラミングと関数型プログラミングの双方のメリットを利用できるバランスの良い言語です。大いに活用しましょう！

参考文献

- Structure and Interpretation of Computer Programs (SICP) 真鍋宏史日本語訳版
- The Swift Programming Language (Swift 2.1)
- Swift Standard Library Reference

Thank you!

2015年11月14日

藤本尚邦 / Cocoa勉強会(関東) #75