函数式编程入门

什么是函数式编程?

Overview

函数式编程 *(functional programming)* ，事实上是一种编程范式 *(programming paradigm)* 。

我们所熟悉的语言种类：面向对象、面向过程，都属于命令式编程 *(imperative programming)* 。而一般认为函数式编程则属于声明式编程 *(declarative programming)* 的一种。

例子：

1. 将一个数组中的所有值平方

   命令式(in Python)

   ```python
   def foo(xs):
     for i in range(len(xs)):
       xs[i] = xs[i] ** 2
     return xs
   ```

   函数式(in Haskell)

   ```haskell
   foo xs = map (^2) xs
   ```

2. 线性素数筛法 ($O(nloglogn)$版本)

   命令式(in Python)

   ```python
   def primes(n):
     P = [True for i in range(n+1)]
     for i in range(2, n+1):
       if P[i]:
         for j in range(i*i, n+1, i):
           P[j] = False
     return [i for i in range(2, n+1) if P[i]]
   ```

   函数式(in Haskell)

   ```haskell
   primes :: Int -> [Int]
   primes n = takeWhile (<n) (filterPrime [2..])
     where filterPrime (p:xs) =
             p : filterPrime [x | x <- xs, x `mod` p /= 0]
   ```

3. 求出所有小于23333的奇平方数的平方和(平方数的平方和)

   命令式(in C++)

```
int bar()
{
    int ans = 0;
    int i = 0;
    while (i++) {
        if (i * i >= 23333) break;
        if (i * i % 2 == 1) ans += i * i * i * i;
    }
    return ans;
}
```

函数式(in Haskell)

```
oddSquareSum = (sum . map (^2) takeWhile (<10000) . filter odd) (map (^2) [1..])
```

他们的主要区别是什么呢?

- 命令式编程关注的是"what"，即"做什么"，它按步骤规定程序要做的事情，并在这一过程中改变状态(计算机的内存、硬盘);
- 声明式编程，或者更确切些，函数式编程关注的是"how"，即"怎么做"，它描述处理数据的方式(定义函数)，组合这些"方式"，来完成对数据的处理;

给出一个比较概括的观点：命令式编程更加贴合机器的结构，状态，流程，都是机器所具有的特征。而函数式编程更加贴合数学与人的抽象思维，更为抽象，符合数学中对问题的思考方式(映射，变换，组合)。

`x = x+1` 与 `f(x) = x+1` :: 命令式编程中有赋值语句(改变状态)，而函数式编程中并没有"赋值"

主要特征

- 函数是头等公民 *(first-class)*： 像值一样被传递、返回

  高阶函数 *(higer-order function)*： 以函数作为参数，或者返回一个函数的函数。

  ```
  compose :: (a -> b) -> (b -> c) -> (a -> c)
  compose f g = \x -> f (g x)
  ```

  ```
  flip :: (a -> b -> c) -> (b -> a -> c)
  flip f = \x y -> f y x
  ```

- 纯函数 *(pure function)*： 无任何副作用的函数

  不依赖于状态，也不改变状态。

  ```
  add1 x = x + 1
  ```

  ```
  x = input()
  print(x)
  ```

- 不可变性 *(immutable)*： 没有"变量"，只有"绑定"。

```
dic = {'a': 1, 'b': 1, 'c': 3}
dic['b'] = b
print(dic)
```

```
(let [m {:a 1 :b 1 :c 3}]
  (println (assoc m :b 2))
  (println m))
```

- 递归 *(recursion)*

  函数式编程的基本概念中，没有"循环"(`for` , `while` , `repeat`)，只有递归。

  有些函数式语言中就算有循环(如Clojure)，也是用递归实现的。

  递归和循环在本质上是等价的，只是是两种不同的思考方式。

  循环：面向机器和具体思维。递归：贴近数学和抽象思维。

  循环能做的事情，递归都能做到，而且往往做的更优雅。

  例如：求n维向量的点乘

  - 循环：

    ```
    def dot(xs, ys):
      n = len(xs)
      prod = 0
      for i in range(n):
        prod += xs[i] * ys[i]
      return prod
    ```

  - 递归：

    ```
    dot :: [Double] -> [Double] -> Double
    dot [] [] = 0.0
    dot (x:xs) (y:ys) = x*y + dot xs ys
    ```

    事实上，利用函数式编程工具箱，可以写成：

    ```
    dot xs ys = sum $ zipWith (*) xs ys
    ```

- Laziness

  Laziness，中译名是"惰性求值"。顾名思义，Laziness或者说惰性求职作为一种语言特性，含义是不到万不得已的时候，并不会求出表达式的值，或者更贴切些，不会对表达式进行求值(表达式带来的副作用也不会发生)。

  Python中对于Laziness的借鉴： `generator`

  对于实现的理解：线段树。当应用函数到值上时，相当于打上了一个标记，等到真正需要的时候再进行计算。

  真正需要的时候：输出到屏幕，写入文件，或者强制求值

函数式编程工具箱

函数式编程工具箱 *(funtional programming toolbox)* ，是一组函数式编程中常用的函数、工具，或者说思维。他们是函数式编程理论中对一些主要问题和方法的抽象，也是一种考虑问题的方法。事实上，他们之间有密不可分的关系，本质上同根同源。

常用函数

他们都其实是不同形式的递归，或者是同一形式递归的不同变体。

1. `map`

   对列表中的所有元素应用一个函数。

   类型签名：

   ```
   map :: (a -> b) -> [a] -> [b]
   ```

   例子：

   ```
   map (+1) [1,2,3] -- [2,3,4]
   ```

   利用递归实现：

   ```
   map _ [] = []
   map f (x:xs) = f x : map f xs
   ```

2. `filter`

   筛选出符合条件的数。

   类型签名：

   ```
   filter :: (a -> Bool) -> [a] -> [a]
   ```

   例子：

   ```
   filter even [1,2,3,4,5]
   ```

   递归实现：

   ```
   filter _ [] = []
   filter p (x:xs)
     | p x = x : filter xs
     | otherwise = filter xs
   ```

3. `fold`

   还有另外一个名字： `reduce` 。

   如你所见，数据科学中的MapReduce，根源便在于函数式编程。也许应用大相径庭，但本质上还是有很多联系。

   `reduce` 这个名字可能更贴切一些，这个函数的作用就是将一个列表 `reduce` 为一个值。这个过程有两个最重要的元素，accumulator和reducer。这是一个迭代的过程，每一次迭代中，accumulator与列表中的一个元素在reducer的作用下产生新的accumulator，进行下一轮迭代。

   听起来可能很复杂，举个具体的例子：用 `fold` 实现求和。

```
fold (+) 0 [1,2,3,4,5]
```

最后的结果是15。

`fold` 的过程如下:

```
5 + 0 => 5
4 + 5 => 9
3 + 9 => 12
2 + 12 => 14
1 + 14 => 15
```

上面的过程中，加号右边的值便是每一轮中的accumulator，加号就是所谓的reducer。

上面还是用"指令式"的方式来理解了这个过程，事实上，`fold` 的过程其实是将列表递归地展开成表达式的过程，比如上面的例子，其实是将列表展开成了如下的表达式:

```
(1 + (2 + (3 + (4 + (5 + 0)))))
```

就像是把一个列表 `fold` 开来了一般。所以从"函数式"的角度来看，似乎是 `fold` 这个名字更为贴切。

`fold` 的类型签名是:

```
fold :: (a -> b -> b) -> b -> [a] -> b
```

事实上，上面提到的 `map` 和 `filter`，都可以用 `fold` 来实现:

```
map :: (a -> b) -> [a] -> [b]
map = foldr (\x acc -> f x : acc) []
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter f = foldr (\x acc -> if f x then x : acc else acc) []
```

> 上面的实现似乎看起来有一点点奇怪—— `foldr`？ `foldr` 是个什么玩意儿？不应该是 `fold` 吗？
>
> 事实上， `fold` 根据实现的方法，有 `foldl` *(fold-left)* 和 `foldr` *(fold-right)* 两种。
>
> 举一个简单的例子说明他们的区别吧：
>
> 对于上面提到的例子 `fold (+) 0 [1,2,3,4,5]`， `foldr` 展开的结果为：
>
> ```
> (1 + (2 + (3 + (4 + (5 + 0)))))
> ```
>
> 而 `foldl` 展开的结果为：
>
> ```
> (((((0 + 1) + 2) + 3) + 4) + 5)
> ```

所以 `fold` 事实上是一类递归模式的抽象：这一类递归作用于列表，边际条件是 `[]`，空列表，每一步处理列表中的首个元素，然后递归地作用到列表的剩余部分。

看一眼 `foldr` 实现方法，就能看清楚这个模式：

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

函数的变换

由于在函数式编程中，函数是"头等公民"，函数也是一种"值"，能被传递，返回，引用。以函数为参数或是返回值的函数，被称为高阶函数。

在函数式编程中，对函数进行的变换也非常常见且使用。

如 `compose` ：

```
compose :: (b -> c) -> (a -> b) -> (a -> c)
compose f g = \x -> f (g x)
```

也就相当于数学中的"复合函数"。

$$(f \circ g)(x) = f(g(x))$$

也有 `flip` ：作用是交换函数的两个参数：

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = \x y -> f y x
```

Recursion + Laziness

之前我们提到了所谓"Laziness"，也即除非万不得已，具有这一特性的编程语言不会求出一个表达式的值。

粗想想，Laziness似乎不会有什么意义，甚至还会让感觉有些恼火。但事实上，Laziness可以成为我们通往"无穷"，操纵"无穷"的一条道路。

> 有Python基础，了解过Generator的同学可能会对此有些了解。

举个例子，我们可以写这样一个函数来返回**所有的**正整数。

```
positives :: [Int]
positives = 1 : map (+1) positives
```

先来检验一下这个函数的正确性吧：他基于这样的一个递归式子：所有自然数组成的序列等于1后面跟上所有自然数序列中每个数加一。

完全正确！

```
[1,2,3, ...] == 1 : [1+1, 2+1, 3+1, ...] == 1 : [2, 3, 4, ...]
```

我们可以尝试在不支持Laziness特性的JavaScript(好吧，严格来说，可能Promise和Laziness有些相通之处)中实现一下这个函数：

```
positives = () =>
  [1].concat
  (
      positives().map(x => x+1)
  )
```

很遗憾，在Javascript中，只要你以任何方式调用 `positives` ，都会形成一个死循环。

好吧，其实Haskell也好不了多少，如果你在Haskell中试图把 `positives` 的结果打印出来，你照样会进入一个死循环。

好消息是：我们有一个安全的办法从 `positives` 中获取数据：

```
take 10 positives -- [1,2,3,4,5,6,7,8,9,10]
```

甚至是这样：

```
let squares = map (^2) positives
take 3 squares -- [1,4,9]
```

也许你已经感觉到了Laziness的威力所在！

我们可以试着探寻一下其中的原理，从 `take` 的实现看起：

```
take :: Int -> [a] -> [a]
take _ [] = []
take 0 _ = []
take n (x:xs) = x : take (n-1) xs
```

接下来想象一下 `take` 作用到 `positives` 上的过程，你可以把这个过程想象为表达式逐步"展开"的过程：

```
take 10 positives -- note: positives = 1 : map (+1) positives
== take 10 (1:map (+1) positives)
== 1 : take 9 (map (+1) positives)
== 1 : take 9 (map (+1) (1 : map (+1) positives))
== 1 : take 9 (2 : map (+2) positives)
== 1 : 2 : take 8 (map (+2) positives)
-- ...
== 1:2:3:4:5:6:7:8:9:10: take 0 (map (+10) positives)
== [1,2,3,4,5,6,7,8,9,10]
```

当然，上面展示的过程的主要目的是示意，在Haskell中实际的实现要更为复杂些。

> Notes:
>
> - `take` 其实也是Lazy的
> - 化简为 `map (+2) positives` 是为了更为简明

More

当然，函数式编程中的思想还远远不止这些。

实际应用中的优劣

优势

- immutable => 带来的安全性，更少的bug
- pure function => 纯函数，便于调试
- 事实上是一种更高效先进的编程范式

  > 如果使用Lisp语言，能让程序变得多短？以Lisp和C的比较为例，我听到的大多数说法是**C代码的长度是Lisp的7倍到10倍**。但是最近，New Architect杂志上有一篇介绍ITA软件公司的文章，里面说"**一行Lisp代码相当于20行C代码**"，因为此文都是引用ITA总裁的话，所以我想这个数字来自ITA的编程实践。 如果真是这样，那么我们可以相信这句话。ITA的软件，不仅使用Lisp语言，还同时大量使用C和C++，所以这是他们的经验谈。

  摘自阮一峰译《黑客与画家》

- ...

劣势

- 更低的执行效率 (immutable)
- "伪函数式" (无副作用等函数式思维较难被人接受)

One Step Further

正如之前提到的，相较而言，函数式编程贴近于"数学"，更为抽象。而事实上，函数式编程的背后隐藏着很多的数学理论，他们是很多函数式编程设计的来源，也是"函数式“的内在根基。

Lambda Calculus & Y-Combinator

说到函数式编程的理论，一定会提到的就是 *Lambda Calculus* ，也就是所谓的"λ演算"。

通常认为，"λ演算"是函数式编程范式最根本的理论基础。因为"λ演算"中强调了函数的重要地位。(事实上，在标准的"λ演算"中，可以说万物皆函数。)

Lambda Calculus是什么

我们先来看看Lambda Calculus长什么样：

$$
\begin{gathered}
\lambda\, x.\, x+1 \\
(\lambda\, x.\, x+1)\ 1 \\
\lambda\, a.\, (\lambda b.\, a+b) \\
(\lambda\, a.\, (\lambda b.\, a+b))\ 1\ 2
\end{gathered}
$$

接下来分行解释一下含义：

1. 定义了一个函数。

2. 将这个函数作用到1，返回2
3. 定义了一个函数，这个函数接收一个参数，返回一个函数。
4. 将这个函数作用到1上，返回一个函数，将返回的函数作用到2上，得到3

简单来说，Lambda Calculus是一种计算模型。

计算模型是什么？还有一种更为有名的计算模型的名字叫做图灵机。确切的说，计算模型是一个用于进行计算的数学模型。他定义了一组符号，以及一组规则，对符号按照规则进行转化，就事实上进行了计算的过程。

> 讲到所谓计算模型，就一定会提到"形式主义"，提到形式主义，就也会讲到哥德尔不完备性定理。这些都是数学中很有趣的问题，但远远超出了讲座的范围之外，有兴趣的同学可以了解一下。

Lambda演算可以不那么严谨的描述为如下：

合法的Lambda表达式有这样几种：

- Variable x
- Lambda Abstraction $\lambda\, x.\, x$
- Lambda Application $(\lambda\, x.\, x)\; y$

Lambda表达式的演算规则主要有两种：

- α替换 $\lambda\, x.3x+1 \to \lambda\, y.\, 3y+1$
- β规约 $(\lambda\, x.3x+1)\; \pi \to 3\pi+1$

柯里化

值得一提的是，Lambda演算中的函数永远只接收一个参数，多参数函数的实现其实在刚刚已经出现过了：

$$\lambda\, a.\, (\lambda\, b.\, a+b)$$

这等价于这样的函数(in JavaScript)：

```
(a, b) => a + b
```

这当然可以写成(in JavaScript)：

```
a => (b => a + b)
```

这事实上被称为柯里化 *(Currying)* ，这样的"多元函数"事实上是这样实现的：接收一个函数，接收第一个参数，然后返回一个函数接收第二个参数。

> 如此实现需要在返回的函数中引用外层函数中的变量，这在很多语言的实现中被称为"词法闭包"。

Y Combinator

Y Combinator，也就是Y组合子，是Lambda Calculus中最有趣也最重要的部分之一。

递归是函数式编程中最重要的基石之一，但也许你发现了，Lambda Calculus中似乎并没有显然的实现递归的方法，因为在这个体系中，并没有所谓"绑定"，也就是说你不能这样做：

```
let factorial n = if n == 0 then 1 else n * factorial (n-1)
```

所以，我们需要Y-Combinator，他如同一个魔术棒，轻轻一挥，我们的函数就能够变成一个递归函数。

他具有这样的形式：

$$Y := \lambda f.(\lambda x.f\,x\,x)\,(\lambda x.f\,x\,x)$$

为什么他具有这样的魔力呢？因为Y组合子具有这样的性质：

$$Y\,f = f\,(Y\,f)$$

他事实上求出了函数的不动点。

那为什么利用函数的不动点可以构造出递归函数呢？

回到上面的问题：求阶乘。我们可以构造这样一个辅助函数：

```
metaFact fact n = if n == 0 then 1 else n * fact (n-1)
```

其中， `fact` 是我们假想出来的那个能实现递归的函数。

> 这可能有些难以理解，但函数式编程中函数本身就是一个值，这其实算是某种意义上的"函数方程" —— `fact` 是一个未知量，而我们试图求解他。这与微分方程有些相似的感觉。

那么，我们可以发现这样的事实： `metaFact fact` 就是我们想要的那个 `fact` ！

> - `metaFact fact` 是什么意思？还记得刚刚讲的柯里化吗？
> - 函数相等的含义是：对于每一个可能的输入，得到的输出都一样。

所以，我们想要的 `fact` 就是 `metaFact` 不动点。

等等，不动点，求不动点这件事情，Y Combinator不是最在行了吗？

所以， `Y metaFact` ，Y组合子发挥了他的魔力。

Type System

Type System，类型系统，指的是编程语言中的类型检查、推导、运算体系。他赋予语言中的量以一些额外的信息——类型。类型系统最主要的目的是避免bug。

Type System的背后是Type Theory，类型论。类型论的体系同样丰富，并且与集合论、范畴论、逻辑学之间有紧密的联系。

> 1. 事实上，根据著名的 *Curry-Howard-Lambek correspondence* （没有找到合适的译名），类型与逻辑之间存在对应关系。我们可以通过构造函数来进行证明。
> 2. 大多数拥有类型系统的函数式语言都对应着直觉主义逻辑。但对Scheme有了解的同学可能知道，Scheme有一些独特之处—— `call/cc` ， `call/cc` 对应于排中律，而直觉主义逻辑并不承认排中律。

函数式编程 $\neq$ 弱类型

似乎在很多人的印象里，函数式语言 $=$ 弱类型语言。这并不完全正确，虽然有Lisp大家族的例子摆着。

事实上，如Idris, Haskell等都拥有一个强大的静态强类型系统，但归功于近乎无所不能的类型推导，可以基本完全脱离类型而写出强类型的代码。事实上，这一类函数式编程语言的类型系统都比C++之流完善健壮得多，在类型推导上C++就不可望其项背，何况还有代数类型等强大特性助阵。

Monad

> 一个单子（Monad）说白了不过就是自函子范畴上的一个幺半群而已。
>
> A monad is just a monoid in the category of endofunctors, what's the problem?
>
> by James Iry in *Brief, Incomplete and Mostly Wrong History of Programming Languages*

关于Monad，似乎这句话大家总是津津乐道，几乎已经成了Haskellers的接头暗号(大雾)。

事实上，这句话最早出现在一部娱乐性比较强的著作里(也就是 *Brief, Incomplete and Mostly Wrong History of Programming Languages*)。

如这句话所暗示的那样，Monad本是一个范畴轮中的概念，随时间发展慢慢被引入到函数式编程中。第一个明确指明Monad与函数式编程之间联系的人是计算机科学家Eugenio Moggi。

事实上，从数学角度来理解Monad是十分不明智的，在函数式编程中，Monad被赋予了一个直观而有意义的主要任务：处理"纯"与"不纯"之间的关系。

之前提到，严格的函数式编程中，事实上是没有"状态"可言的。一个纯函数，不管什么时候调用他，只要参数给定，那么得到的结果一定是相等的。我们不能赋值，不能改变"状态"，也不能读取"状态"。

但是打印字符需要状态，读取键盘输入需要状态，产生随机数需要状态，如果完全扼杀状态的存在，函数式编程将在实际应用中寸步难行。

于是，便到了Monad发挥作用的时候。Monad的本质，事实上是一个"运算描述"。举个例子，如果我们需要写一个函数打印"Hello, world"，我们并不直接在函数中实现这件事情，而是让这个函数返回一个Monad——一个运算描述，里面写着："我要打印'Hello, world'"。这样，我们的函数仍然是"纯"的，那些涉及状态的事情，便在对这个Monad求值的时候完成。

举个最简单的例子：

```haskell
main :: IO ()
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn "Hello, " ++ name
```

可以看到， `main` 的类型签名中返回的 `IO ()` 就是一种Monad，也即IO Monad。而 `do` 就像一个胶水函数，他的作用是把多个Monad连接起来，也就是把几个"运算描述"整合成一个。

所以我们可以这样理解，我们的程序就是一个大大的Monad，运行我们程序所做的事情就是对这个Monad求值，也就完成了Monad所描述的运算。而我们的Haskell代码仍然是"纯"的，他做的事情只是返回了这个描述而已。

Haskell中的理论体现

为什么在讲座的最后，要提及这样的一个点呢？因为这件事情真的值得一提。

Haskell是由一群计算机科学理论的博士设计的，他设计的初衷是整合当时所有的函数式语言，作为以后在函数式编程理论开展研究的基础。

所以，说白了，Haskell设计的初衷就是一个玩具，一个试验场，也注定了他的血液里流淌的不是工程实践，而是数学理论。

> 事实上，在数十年的发展之后，看起来华而不实的锈花刀在实战中也不比厚重的大砍刀逊色。

Lambda Calculus & Currying

事实上，严格来说，Haskell中的函数都只接受一个参数，他们都是柯里化的函数。还记得Haskell函数的类型签名吗？

```
add :: Int -> Int -> Int
add x y = x + y
```

所以我们可以做这样的事情：

```
addOne :: Int -> Int
addOne = add 1
```

事实上， `add 1` 就等价于：

```
add y = 1 + y
```

所以，类型签名其实等价于：

```
add :: Int -> (Int -> Int)
```

Type System

Haskell中有极为完备的类型系统。

如非常常用且强大的的代数数据类型 *(Algebraic data type)* 。

代数数据类型中的"代数"的含义为，这样的数据类型是由代数运算构造出来的：

- Sum :: 和
- Product :: 积

最简单的例子：

1. 和运算

   ```
   data Boolean = True | False
   ```

   和的意思其实类似于"或"。这里用 `True` 和 `False` 两个常量作和构造出了 `Boolean` 类型。

2. 积运算

```
data Point = P Double Double
```

这里的 `P Double Double` 就是积运算。"积"的含义类似于且。积运算构造出的类型在产生示例的时候，必须给出所有成员的示例。比如 `P 1.0 1.0` 。

当然我们可以做更多复杂但有趣的事情：

定义我们的链表：

```
data List a = EmptyList | Cons a (List a)
```

定义我们的二叉树：

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a)
```

> 事实上，Haskell的大多数内建数据结构，如列表，布尔值。都是一种代数数据类型。只不过有关他们的所有构造函数与运算符都是内置 *(built-in)* 的。

> 事实上，Haskell甚至可以成为类型论推导的工具。

Monad

Haskell的标准库中就实现了 `Monad` Type Class，以及很多相关的函数。事实上，正如之前所提到的，Haskell与现实世界中的状态打交道的主要途径就是 `Monad` 。

...

这并不是结束。