函数式编程 (FP) 入门

以 Javascript 为例

Karl Xiao

Carnegie Mellon University

2022 年 1 月 16 日

目录

FP 的定义及历史

什么是 FP?

函数式编程 (functional programming, 以下简称 FP) 是一种把函数 (functions) 放在首位，更接近于数学的编程风格。它是一种与指令式编程 (imperative programming) 相对应的理念，并不是一门/一类编程语言。

FP 简史

FP 的理论基础为 1936 年由数学家阿隆佐 · 邱奇 (Alonzo Church) 创建的 λ 演算 (λ-Calculus) 系统。同时，FP 的一些概念来自 40 年代发明的极为抽象的范畴论 (category theory)。

FP 的语言支持

不同语言对 FP 有不同程度上的支持：

基本不支持	C
少量支持	C++, Go, Java
部分支持	Javascript, Python, Rust
"FP 语言"	Scala, F#, OCaml, SML
" 纯 FP"	Haskell

本讲义将以 Javascript 为例。Javascript 的语法较为熟悉，但仍保留 FP 的精髓。

头等函数 (First-class Functions)

进行 FP 需要的是函数被当作头等公民 的语言。这意味着任意函数可以：

- 作为其它函数的参数
- 作为其它函数的返回值
- 存于拥有函数类型 (e.g. `int -> int`) 的普通变量
- 作为匿名函数 (anonymous function) 存在

于是，FP 主要通过函数的巧妙叠加进行运算。这些运算与普通数学运算相似。

Example

数学中的 $f(x) = x + 1$ 在不同编程语言中对应的函数变量：

- `let f = fun x -> x + 1` (OCaml)
- `let f = x => x + 1` (Javascript)
- `f = lambda x:  x + 1` (Python)
- `auto f = [&](int x){ return x + 1; }` (C++)

高阶函数 (Higher Order Function)

我们可以在支持 FP 的语言中创建满足以下条件的高阶函数:

- 接受一个或多个函数作为输入
- 输出一个函数

需要强调的是，高阶函数不仅在 FP 里常常用到，作为算子 $(\Delta, \nabla, \nabla\times, \mathcal{L}, A\ldots)$ 在数学中也处处可见。

Example

(提示：微积分)

高阶函数 (Higher Order Function)

我们可以在支持 FP 的语言中创建满足以下条件的高阶函数:

- 接受一个或多个函数作为输入
- 输出一个函数

需要强调的是，高阶函数不仅在 FP 里常常用到，作为算子 $(\Delta, \nabla, \nabla\times, \mathcal{L}, A\dots)$ 在数学中也处处可见。

Example

(不定积分) $I : \mathcal{C}([0,1]) \to \mathcal{C}([0,1])$

$$f \mapsto If$$

$$[If](x) := \int_0^x f(t)\,dt$$

柯里化 (Currying)

Definition

柯里化 (Currying) 以逻辑学家 Haskell Curry 命名，指的将接受多个参数的函数变换成 [接受一个单一参数（最初函数的第一个参数），返回另一个函数] 的函数的技巧。

Example

```
1   // curry 前: int * int -> int
2   function addTwoNumbers(x,y) {
3       return x + y;
4   }
5   const addTwoNumbers = (x,y)=>x+y; // 更精简的写法
6
7   // curry 后: int -> (int -> int)
8   function add_curried(x) {
9       return function(y) {
10          return x + y;
11      }
12  }
13  const add_curried = x=>(y=>x + y); // 更精简的写法
14
15  // 示例
16  const add3 = add_curried(3); // add3 仍为函数
17  console.log(add3(4) == addTwoNumbers(3,4));
```

函数“工厂”

Currying 使我们能够用短短几行代码换来无穷多个函数。同时，我们可以把复杂的运算分解成两个或多个步骤。

Example

```
1   // 问题：以下curry函数的（多态性）类型是...？
2   const curry = f=>(x=>y=>f(x,y));
3
4   const add_curried = curry(addTwoNumbers);
5   const add_custom = add_curried(prompt("输入任何数字"));
6   // 这里add_custom可以是add1、add2、add666等
7
8   //  int -> (int -> bool)
9   const checkPrimeN = N=>{
10      let table = new Array(N);
11
12      // 往table的每个索引填true或者false（省略）
13
14      return (x=>table[x]); // 函数闭包
15  }
16  const checkPrime1000 = checkPrimeN(50); // int -> bool
17
18  // 接下来就可以快速(O(1))检索任意小于1000的数字是否为质数
19  console.log(checkPrime(957)) // false
```

函数闭包 (Closure)

闭包是一个函数值，它引用了其函数体之外的变量。该函数可以访问并赋予其引用的变量的值，换句话说，该函数被这些变量“绑定”在一起。

Example

```
1   let a0 = -1;
2   let a1 = -2;
3   const f = ()=>{
4       let a0 = 0;
5       let a1 = 1;
6       return ()=>{
7           let ret = a0;
8           a0 = a1;
9           a1 = ret + a0;
10          return ret;
11      }
12  }
13
14  const fib = f();
15  for (let i = 0; i < 10; i++) {
16      console.log(fib()); // 问题(1): 会打印什么?
17  }
18
19  console.log(a0, a1); // 问题(2): 在这里a0 = ?, a1 = ?
```

纯 FP

Definition

纯 FP 禁用状态变更 (state/ side-effects) 和可变 (mutable) 数据。所有数据结构 (DS) 都是不可变的 (immutable)。所有函数的运算结果都只能由参数决定。

Example

纯 FP 禁用数组 (array)、全局变量 (global variable) 以及大多数的哈希表 (hash table)，但可以（而且主要）用串列 (list) 和二叉查找树 (binary search tree)。

递归 (Recursion)

注意

纯 FP 不允许使用 for/while 循环，尽可能用递归替代。

Example

```
1       // 阶乘函数，假设N >= 0
2       const fact = N=>{
3           let res = 1;
4           for (let i = 1; i<=N; i++) {
5               res *= i;
6           }
7           return res;
8       };
9
10      // 递归（注：可以使用"尾递归"优化）
11      const fact = N=>(N<=1 ? 1 : N * fact(N-1));
12
13      // 引申（x:基线条件，f:"适当类型"的函数）
14      // 例：loop((x,y) => x * y)(1) == fact
15      const loop = f=>x=>N=>(N == 0 ? x : f(N, loop(f)(x)(N-1)));
```

常用高阶函数

以下函数组建新的 DS，不改变已有的 DS。

- `map`
- `filter`
- `reduce`
- `flat`

Example

```
1    const array1 = [1,2,3,4];
2    const array2 = array1.map(x=>2*x) // [2,4,6,8]
3    const array3 = array1.filter(x=>x % 2 == 0) // [2,4]
4    const array4 = array1.reduce((x,y)=>x + y) // 10
5
6    const array5 = [[1,2], [3,4]];
7    const array6 = array5.flat(); // [1,2,3,4]
```

注意

这些函数在 Javascript 里是 Array 的方法，不是真正的高阶函数。比方说，OCaml 里相应的的 map 函数的类型为 `val map : ('a -> 'b) -> 'a list -> 'b list`。此外，JavaScript 的 Array 其实是可变的 (但 list 不可变)。

FP 的优点

- 代码更为简洁
- 不容易出错 (减少 bug!)
- 适用于多线程/并行计算 (分布式、MapReduce、Spark...)
- 可进行自动化定理证明 (Coq、四色定理...), 柯里-霍华德同构 (Curry-Howard Correspondence), 构造逻辑 (Constructive Logic) ...
- ...

其他重要概念

- 多态性 (Polymorphism)
- 代数数据类型 (Algebraic Data Types)
- 模式匹配 (Pattern-Matching)
- 尾递归 (Tail recursion)
- 后续传递风格 (Continuous Passing Style)
- 单子 (Monad)
- ...

一些链接

- 官方 Ocaml 教程
- 官方 React 教程
- Real World Haskell 中文版
- ...