函数式编程入门

> 施程航 18/10/24

啥是函数式编程

函数编程语言 - 维基百科 。函数式语言是一种独特的编程范式。简单来说，就是把函数的地位与变量等同，我们可以像操作、组合变量那样折腾函数。这里的函数可以直接认为是数学上的映射。函数式编程语言也有很多，可以按照不同的标准分为不同的类型，从动态类型的到静态类型的，或者从及早求值（*Eager Evaluation*）的到惰性求值的（*Lazy Evaluation*）。

这里我们选用的语言是 Scheme，Lisp 语言的变种。Lisp 语言是像 Python 一样的动态类型语言，以「丰富」的括号闻名。至于这么多括号的意义，到下面就会明白。对于 Lisp 的运行环境，可以参考 课前准备 ，用 Racket、GNU Guile、MIT Scheme 或者在线的解释器都可以。

开始之前

让我们回顾下列表、序对和递归的概念。所谓 Lisp，就是 List Processor 的缩写，也就是说，Lisp 语言的核心构造之一就是列表——广义上，你看到的所有括号里以空格隔开的符号，都构成了一个列表。所谓序对，就是一个二元组，比如 `(1, 2)` 。那么序对在 Lisp 中存在于何处呢？它和列表又有什么关系呢？

要讨论这个问题，我们首先要学会在 Lisp 里面操纵它们。

```
;; 我相信你已经会输出东西了
(display "Hello, world!")

;; 所以你应该明白了，在 Lisp 里面，最基本的一条语法（本来也为数不多）就是
;; (技能名称 施法对象1 施法对象2 施法对象3 ...)
;; 还有就是注释用英文分号开头！

;; 来创建一个序对吧
(cons 1 2)

;; 访问第一个
(car (cons 1 2)) ;; => 1

;; 访问第二个
;; car 和 cdr 的名字属于历史问题，感兴趣可以搜索一下，和车没关系啦！
(cdr (cons 1 2)) ;; => 2

;; 这是列表啦
(list 1 2 3 4 5)

;; 可以用 car 访问列表的第一个元素吗？
(car (list 1 2 3 4 5)) ;; => 1

;; 用 cdr 会得到 2 吗？
;; 好像不太对劲
(cdr (list 1 2 3 4 5)) ;; => '(2 3 4 5)
(cdr (cdr (list 1 2 3 4 5))) ;; => '(3 4 5)
(cdr (list 5)) ;; => '()
```

```
(cdr (list)) ;; => Boom! 报错了
```

虽然 `cdr` 的结果和想象中可能有些不一样，不过如果忽略符号表示上的一些小差异，你会发现，所谓的 `(1, 2, 3, 4, 5)` 其实是 `(1, (2, (3, (4, 5))))` 。

至于递归嘛，参考 `fibnacci` 函数。

数据的映射

比如，为了得到一个翻转的序对， `(1,2)` 到 `(2,1)` ，函数式点的做法是：

```
;; 不难看出，这段代码用 x 的 cdr 和 car 新建了一个序对
(define (_reverse x)
    (cons (cdr x) (car x))
)

;; 让我们测试下
(define a (cons 1 2))
(define b (_reverse a))

;; 打印 a, b
(display a) ;;(1, 2)
(newline)
(display b) ;;(2, 1)
(newline)
```

也就是说，我们在意的是数据到数据的映射，而不是改变原来的数据，这与数学上的函数相呼应。这里的「不变」可以带来一些好处，后面会提到。为何在意「不变」？如果翻译成 C 语言，这段代码应该像这样：

```
/* Caution: This is not correct code. */

int* reverse(int arr[2]) {
    int result[2] = { arr[1], arr[0] };
    return result;
}

int main(void) {
    int arr[2] = { 1, 2 };
    int* another = reverse(arr);
    return 0;
}
```

这代码当然是有问题的（不代表运行就会崩溃），为了让它能通过编译甚至还得把 `reverse` 的返回类型从 `int[2]` 改成 `int*` 。不过我想你应该明白这意思了——「不改变任何现有的变量值」是一种崭新的编程手法（尽管在 Lisp 里你也可以改）。你甚至会惊奇地发现——就算每个变量的值都不许改，你依然能写出表达能力一样强的程序！老实说，当你第一次接触到 `x = x + 1` 这种 C 语言语句的时候，有没有觉得别扭？承认吧，不可变才更符合数学思维……

高阶函数

好了，这里我们又引进了一个新概念，概念总是无聊的：

> 高阶函数就是以函数为参数或者返回函数的函数

有点拗口？这里就是前面说的把函数的地位与变量等同。

> Show me the code?

- 求和函数

```
;; 比如我们想要求两数/两数平方/两数立方之和
;; 我们可能会这样干
;; 两数之和
(define (sum-of-self x y)
    (+ x y)
)

;; 平方
(define (sum-of-square x y)
    (+ (* x x) (* y y))
)
;; 立方
(define (sum-of-cube x y)
    (+ (* x x x) (* y y y))
)

;; 我是可爱的分割线

;; 我们可以发现这三个函数内部实际上都是：
;; (+ (f x) (f y))
;; 所以我们也可以这样搞
(define (self x) x) ;; 这里可能会有疑问
(define (square x) (* x x))
(define (cube x)(* x x x))

;;
(define (sum x y f)
    (+ (f x) (f y))
)

;;
(define (sum-of-self x y)
    (sum x y self)
)
(define (sum-of-square x y)
    (sum x y square)
)
(define (sum-of-cube x y)
    (sum x y cube)
)
```

测试代码：

```
(display (sum-of-self 3 2))
(newline)
(display (sum-of-square 3 2))
(newline)
(display (sum-of-cube 3 2))
(newline)
```

你可能会说，代码变长了，的确。不过，让我们看看我们是怎样把代码“变长”的：提取公共模式，每个 `sum` 函数内部都有 `(+ (f x) (f y))` 的形式，把这部分抽出来，用 `f` 去代替 `self square cube` ，我们得到能够代表三个求和函数的形式 `sum` ，它有一个参数 `f` ，它表示我们如何转化 `x` `y` 。

- `filter` （表的过滤） OK，缓一缓，你可能觉得这个求和函数意义还没那么大。我们来想象一个场景：给你一列成绩，如何拿到里面及格的所有成绩呢？我们先用 C 语言写一个：

```c
int main(void) {
    int arr[] = { 56, 78, 92, 99, 17, 83, 65, 37 };
    int len = 8;
    int result[8] = {};
    int result_len = 0;
    for (int i = 0; i < len; ++i) {
        if (arr[i] > 60) {
            result[result_len++] = arr[i];
        }
    }
    return 0;
}
```

改写成一个函数？我想你也会。不过如果我需要写一个通用的函数，不管及格的分数呢？好的，你会说，给函数加上一个参数 `standard` 表示及格线的分数就可以了。那如果我突然有一天说，这个要结合学生的历史成绩，而历史成绩在另一个函数里……解决方法就是把「判断及格」这个代码作为一个单独的函数，然后把「判断及格」的函数作为我这个通用函数的参数。好了，这个通用函数就是 `filter` 。而这个「判断及格」就是 `predicate` 。 `filter` 的定义类似：

```lisp
(define (filter predicate sequence)
    ;; balabala...
)
```

- `map` （表到表的映射）

`map` 和判断成绩是否及格的 `filter` 有些相似，不过用在老师们想多让一些同学们过的场合，也就是把一个列表中的每个元素都通过 `transform` 函数转换一遍然后返回新的列表。 `map` 的定义类似：

```lisp
(define (map transform sequence)
    ;; ...
)
```

```lisp
;; 这里把列表里的数转成了它们的相反数，然后组成一个新的列表

(map - (list 1 2 3 4)) ;; -> (-1 -2 -3 -4)
;
(define (bigger-than-0 x)
    (> x 0))
(filter bigger-than-0 (list -1 2 4 -2 0 3))
```

```lisp
(define (promote-scores seq)
  (map (lambda (x) (* (sqrt x) 10)) seq))
(promote-scores (list 56 78 92 99 17 83 65 37))

```

```
;; => '(74.83314773547883
;;  88.31760866327848
;;  95.91663046625439
;;  99.498743710662
;;  41.23105625617661
;;  91.10433579144299
;;  80.62257748298549
;;  60.82762530298219)
;;
;; 有两个本来没过的现在过了
```

在静态类型的函数式语言里还存在一个类似的高阶函数叫做 `flatMap` ，不要手贱去搜，会烧脑的……像这样的高阶函数还有很多，接触多了就会发现没有什么神奇的，比如你自己也可以发明一个函数叫做 `reject` ，作用就是返回所有不满足 `predicate` 的元素。

- 手写快排？ 快排是啥？自行搜索「快速排序」。（放心，这四个字会伴随你们很久的……）

```
;; 本地环境的同学，把右下角的语言换成“大”

(define (qsort seq)
    (if (null? seq)
        '()
        (append
        (qsort (filter (lambda (x) (< x (car seq))) seq))
        (filter (lambda (x) (= x (car seq))) seq)
        (qsort (filter (lambda (x) (> x (car seq))) seq))
        )
    ))
(qsort (list 2 1 4 3 -1))
```

> 如果我们想降序/非升序排列呢…

> 如果我们想按照其他方式排列呢…

问题来了，借一步说话！

我是写 C/C++/Python/... 的，函数式可以干嘛

函数指针、仿函数...这里还是拿快排做例子，用C试试？可以在 [这里](#) 跑下下面的代码

```
这是qsort的声明
void qsort(void *base,//数组的第一个元素的指针
           size_t nitems,//元素个数
           size_t size,//元素占内存的大小
           int (*comp)(const void *, const void*)//比较元素大小的函数
       );
```

```
// ps:这里看不懂不要紧，因为指针这部分是有、难...知道大概的想法就Ok(吐槽，这里指针的语法真的丑...)
#include <stdio.h>
#include <stdlib.h>

int values[] = {3, 6, 7, 1, 2, 3,10};
int cmp1(const void* a, const void *b)
```

```
{
    return (*(int*)a - *(int*)b);
}
int cmp2(const void* a, const void *b)
{
    int _a = *(int*)a, _b = *(int*)b;
    if(_a%3 == _b%3)
        return _a - _b;
    else
        return (_a%3 - _b%3);
}

int main()
{
    qsort(values, 7, sizeof(int), cmp1);
    int n;
    printf("第一次排序:\n");
    for(n = 0; n < 7;n++)
    {
        printf("%d  ",values[n]);
    }
    printf("\n");

    //我是超可爱的分割线


    qsort(values, 7, sizeof(int), cmp2);
    printf("第二次排序:\n");
    for(n = 0; n < 7;n++)
    {
        printf("%d  ",values[n]);
    }
    printf("\n");

    return 0;
}
```

C++ 当然也对高阶函数有很好的支持。

```
#include <algorithm>

struct PlusOp {
    int operator () (int x) {
        return x + 2;
    }
};

int main(void) {
    int source[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int target[10] = {};
    PlusOp op;
    std::transform(source, source + 10, target, op);
    return 0;
}
```

对于 Python，有兴趣的可以看看 [廖雪峰老师 Python课程中的函数式编程部分]() 。

接触不同的语言和思想，有利于打开自己的编程视野。事实上，绝大多数现代编程语言都具备了对高阶函数等函数式语言特性的支持，包括但不限于 Java、C++、Swift、Rust、Scala、Python、Ruby、PHP、JavaScript、C#、Go……

- # 拓展：延时求值

```
;; 写个从n开始计数的序列

(define (counter-from n)
    (cons n (counter-from (+ n 1)))
)
(display (counter-from 1))
(display "来追我呀！")

;; 看看发生了啥
```

> 延时求值，就是需要一个表达式的值时再去求值这个表达式

- # 总结与希望
  - 语言和范式影响思考的方式
  - 类似的思想存在于相同的语言
- # 欢迎大家下次、下下次、下下下次...来蹭大佬们开的课！