

函数式编程初探

作者：[阮一峰](#)

诞生50多年之后，[函数式编程](#)（functional programming）开始获得越来越多的关注。

不仅最古老的函数式语言Lisp重获青春，而且新的函数式语言层出不穷，比如Erlang、clojure、Scala、F#等等。目前最当红的Python、Ruby、Javascript，对函数式编程的支持都很强，就连老牌的面向对象的Java、面向过程的PHP，都忙不迭地加入对匿名函数的支持。越来越多的迹象表明，函数式编程已经不再是学术界的最爱，开始大踏步地在业界投入实用。

也许继"面向对象编程"之后，"函数式编程"会成为下一个编程的主流范式（paradigm）。未来的程序员恐怕或多或少都必须懂一点。

$f(x)$

但是，"函数式编程"看上去比较难，缺乏通俗的入门教程，各种介绍文章都充斥着数学符号和专用术语，让人读了如坠云雾。就连最基本的问题"什么是函数式编程"，网上都搜不到易懂的回答。

下面是我的"函数式编程"学习笔记，分享出来，与大家一起探讨。内容不涉及数学（我也不懂[Lambda Calculus](#)），也不涉及高级特性（比如[lazy evaluation](#)和[currying](#)），只求尽量简单通俗地整理和表达，我现在所理解的"函数式编程"以及它的意义。

我主要参考了Slava Akhmechet的["Functional Programming For The Rest of Us"](#)。

一、定义

简单说，"函数式编程"是一种["编程范式"](#)（programming paradigm），也就是如何编写程序的方法论。

它属于["结构化编程"](#)的一种，主要思想是把运算过程尽量写成一系列嵌套的函数调用。举例来说，现在有这样一个数学表达式：

```
(1 + 2) * 3 - 4
```

传统的过程式编程，可能这样写：

```
var a = 1 + 2;
```

```
var b = a * 3;

var c = b - 4;
```

函数式编程要求使用函数，我们可以把运算过程[定义](#)为不同的函数，然后写成下面这样：

```
var result = subtract(multiply(add(1,2), 3), 4);
```

这就是函数式编程。

二、特点

函数式编程具有五个鲜明的特点。

1. 函数是"第一等公民"

所谓"[第一等公民](#)" (first class)，指的是函数与其他数据类型一样，处于平等地位，可以赋值给其他变量，也可以作为参数，传入另一个函数，或者作为别的函数的返回值。

举例来说，下面代码中的print变量就是一个函数，可以作为另一个函数的参数。

```
var print = function(i){ console.log(i);}

[1,2,3].forEach(print);
```

2. 只用"表达式"，不用"语句"

"表达式" (expression) 是一个单纯的运算过程，总是有返回值；"语句" (statement) 是执行某种操作，没有返回值。函数式编程要求，只使用表达式，不使用语句。也就是说，每一步都是单纯的运算，而且都有返回值。

原因是函数式编程的开发动机，一开始就是为了处理运算 (computation)，不考虑系统的读写 (I/O) 。“语句”属于对系统的读写操作，所以就被排斥在外。

当然，实际应用中，不做I/O是不可能的。因此，编程过程中，函数式编程只要求把I/O限制到最小，不要有不必要的读写行为，保持计算过程的单纯性。

3. 没有"副作用"

所谓"[副作用](#)" (side effect)，指的是函数内部与外部互动（最典型的情况，就是修改全局变量的值），产生运算以外的其他结果。

函数式编程强调没有"副作用"，意味着函数要保持独立，所有功能就是返回一个新的值，没有其他行为，尤其是不得修改外部变量的值。

4. 不修改状态

上一点已经提到，函数式编程只是返回新的值，不修改系统变量。因此，不修改变量，也是它的一个重要特点。

在其他类型的语言中，变量往往用来保存"状态" (state)。不修改变量，意味着状态不能保存在变量中。函数式编程使用参数保存状态，最好的例子就是递归。下面的代码是一个将字符串逆序排列的函数，它演示了不同的参数如何决定了运算所处的"状态"。

```
function reverse(string) {

    if(string.length == 0) {
```

```

        return string;
    } else {
        return reverse(string.substring(1, string.length)) + string.substring(0, 1);
    }
}

```

由于使用了递归，函数式语言的运行速度比较慢，这是它长期不能在业界推广的主要原因。

5. 引用透明

引用透明（Referential transparency），指的是函数的运行不依赖于外部变量或"状态"，只依赖于输入的参数，任何时候只要参数相同，引用函数所得到的返回值总是相同的。

有了前面的第三点和第四点，这点是很显然的。其他类型的语言，函数的返回值往往与系统状态有关，不同的状态之下，返回值是不一样的。这就叫"引用不透明"，很不利于观察和理解程序的行为。

三、意义

函数式编程到底有什么好处，为什么会变得越来越流行？

1. 代码简洁，开发快速

函数式编程大量使用函数，减少了代码的重复，因此程序比较短，开发速度较快。

Paul Graham在《[黑客与画家](#)》一书中[写道](#)：同样功能的程序，极端情况下，Lisp代码的长度可能是C代码的二十分之一。

如果程序员每天所写的代码行数基本相同，这就意味着，"C语言需要一年时间完成开发某个功能，Lisp语言只需要不到三星期。反过来说，如果某个新功能，Lisp语言完成开发需要三个月，C语言需要写五年。"当然，这样的对比故意夸大了差异，但是"在一个高度竞争的市场中，即使开发速度只相差两三倍，也足以使得你永远处在落后的位置。"

2. 接近自然语言，易于理解

函数式编程的自由度很高，可以写出很接近自然语言的代码。

前文曾经将表达式 $(1 + 2) * 3 - 4$ ，写成函数式语言：

```
subtract(multiply(add(1,2), 3), 4)
```

对它进行变形，不难得到另一种写法：

```
add(1,2).multiply(3).subtract(4)
```

这基本就是自然语言的表达了。再看下面的代码，大家应该一眼就能明白它的意思吧：

```
merge([1,2],[3,4]).sort().search("2")
```

因此，函数式编程的代码更容易理解。

3. 更方便的代码管理

函数式编程不依赖、也不会改变外界的状态，只要给定输入参数，返回的结果必定相同。因此，每一个函数都可以被看做独立单元，很有利于进行单元测试（unit testing）和除错（debugging），以及模块化组合。

4. 易于"并发编程"

函数式编程不需要考虑"死锁" (deadlock) , 因为它不修改变量, 所以根本不存在"锁"线程的问题。不必担心一个线程的数据, 被另一个线程修改, 所以可以很放心地把工作分摊到多个线程, 部署"并发编程" (concurrency) 。

请看下面的代码:

```
var s1 = Op1();  
  
var s2 = Op2();  
  
var s3 = concat(s1, s2);
```

由于s1和s2互不干扰, 不会修改变量, 谁先执行是无所谓的, 所以可以放心地增加线程, 把它们分配在两个线程上完成。其他类型的语言就做不到这一点, 因为s1可能会修改系统状态, 而s2可能会用到这些状态, 所以必须保证s2在s1之后运行, 自然也就不能部署到其他线程上了。

多核CPU是将来的潮流, 所以函数式编程的这个特性非常重要。

5. 代码的热升级

函数式编程没有副作用, 只要保证接口不变, 内部实现是外部无关的。所以, 可以在运行状态下直接升级代码, 不需要重启, 也不需要停机。[Erlang](#)语言早就证明了这一点, 它是瑞典爱立信公司为了管理电话系统而开发的, 电话系统的升级当然是不能停机的。