编程范式游记（4）– 函数式编程

2018-01-23 陈皓

从前三章内容中，我们了解到，虽然 C 语言简单灵活，能够让程序员在高级语言特性之上轻松进行底层上的微观控制，被誉为“高级语言中的汇编语言”，但其基于过程和底层的设计初衷又成了它的短板。

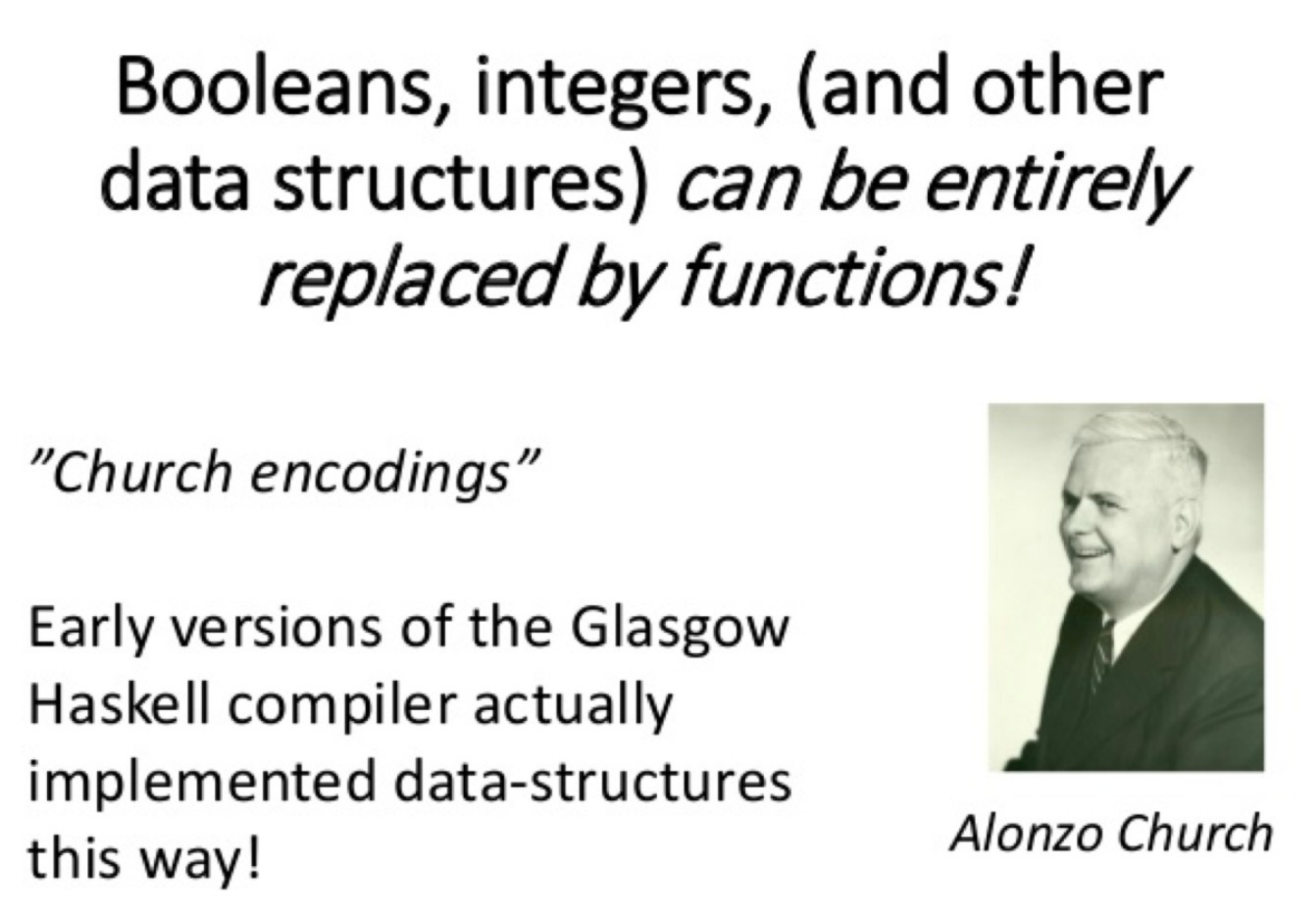
在程序世界中，编程工作更多的是解决业务上的问题，而不是计算机的问题，我们需要更为贴近业务更为抽象的语言，如面向对象语言 C++ 和 Java 等。

C++ 很大程度上解决了 C 语言中的各种问题和不便，尤其是通过类、模板、虚函数和运行时识别等解决了 C 语言的泛型编程问题。然而，如何做更为抽象的泛型呢？答案就是函数式编程（Functional Programming）。

函数式编程

相对于计算机的历史而言，函数式编程其实是一个非常古老的概念。函数式编程的基础模型来源于 λ 演算，而 λ 演算并非设计于在计算机上执行。它是由 Alonzo Church 和 Stephen Cole Kleene 在 20 世纪 30 年代引入的一套用于研究函数定义、函数应用和递归的形式系统。

如 Alonzo 所说，像 booleans、integers 或者其他的数据结构都可以被函数取代掉。



Booleans, integers, (and other data structures) *can be entirely replaced by functions!*

”Church encodings”

Early versions of the Glasgow Haskell compiler actually implemented data-structures this way!

Alonzo Church

我们来看一下函数式编程，它的理念就是借用于数学中的代数。

```
f(x)=5x^2+4x+3
g(x)=2f(x)+5=10x^2+8x+11
h(x)=f(x)+g(x)=15x^2+12x+14
```

假设 f(x) 是一个函数，g(x) 是第二个函数，把 f(x) 这个函数套下来，并展开。然后还可以定义一个由两个一元函数组合成的二元函数。还可以做递归，下面这个函数定义就是斐波那契数列。

```
f(x)=f(x-1)+f(x-2)
```

对于函数式编程来说，其只关心，**定义输入数据和输出数据相关的关系，数学表达式里面其实是在做一种映射（mapping），输入的数据和输出的数据关系是什么样的，是用函数来定义的。**

函数式编程有以下特点。

特征

- **stateless**：函数不维护任何状态。函数式编程的核心精神是 stateless，简而言之就是它不能存在状态，你给我数据我处理完扔出来，里面的数据是不变的。
- **immutable**：输入数据是不能动的，动了输入数据就有危险，所以要返回新的数据集。

优势

- 没有状态就没有伤害。
- 并行执行无伤害。
- Copy-Paste 重构代码无伤害。
- 函数的执行没有顺序上的问题。

函数式编程还带来了以下一些好处。

- **惰性求值**。这需要编译器的支持。表达式不在它被绑定到变量之后就立即求值，而是在该值被取用的时候求值。也就是说，语句如 `x:=expression;`（把一个表达式的结果赋值给一个变量）显式地调用这个表达式被计算并把结果放置到 `x` 中，但是先不管实际在 `x` 中的是什么，直到通过后面的表达式中到 `x` 的引用而有了对它的值的需求的时候，而后面表达式自身的求值也可以被延迟，最终为了生成让外界看到的某个符号而计算这个快速增长的依赖树。

- **确定性**。所谓确定性，就是像在数学中那样，`f(x) = y` 这个函数无论在什么场景下，都会得到同样的结果，这个我们称之为函数的确定性。而不是像程序中的很多函数那样，同一个参数，却会在不同的场景下计算出不同的结果。所谓不同的场景，就是我们的函数会根据运行中的状态信息的不同而发生变化。

我们知道，因为状态，在并行执行和 copy-paste 时引发 bug 的概率是非常高的，所以没有状态就没有伤害，就像没有依赖就没有伤害一样，并行执行无伤害，copy 代码无伤害，因为没有状态，代码怎样拷都行。

劣势

- 数据复制比较严重。

> **注**：有一些人可能会觉得这会对性能造成影响。其实，这个劣势并不见得会导致性能不好。因为没有状态，所以代码在并行上根本不需要锁（不需要对状态修改的锁），所以可以拼命地并发，反而可以让性能很不错。比如：Erlang 就是其中的代表。

对于纯函数式（也就是完全没有状态的函数）的编程来说，各个语言支持的程度如下。

- 完全纯函数式 Haskell
- 容易写纯函数 F#, Ocaml, Clojure, Scala
- 纯函数需要花点精力 C#, Java, JavaScript

完全纯函数的语言，很容易写成函数，纯函数需要花精力。只要所谓的纯函数的问题，传进来的数据不改，改完的东西复制一份拷出去，然后没有状态。

但是很多人并不习惯函数式编程，因为函数式编程和过程式编程的思维方式完全不一样。过程式编程是在把具体的流程描述出来，所以可以不假思索，而函数式编程的抽象度更大，在实现方式上，函数套函数，函数返回函数，函数里定义函数……把人搞得很糊涂。

函数式编程用到的技术

下面是函数式编程用到的一些技术。

- **first class function（头等函数）**：这个技术可以让你的函数就像变量一样来使用。也就是说，你的函数可以像变量一样被创建、修改，并当成变量一样传递、返回，或是在函数中嵌套函数。

- **tail recursion optimization（尾递归优化）**：我们知道递归的害处，那就是如果递归很深的话，stack 受不了，并会导致性能大幅度下降。因此，我们使用尾递归优化技术——每次递归时都会重用 stack，这样能够提升性能。当然，这需要语言或编译器的支持。Python 就不支持。

- **map & reduce**：这个技术不用多说了，函数式编程最常见的技术就是对一个集合做 Map 和 Reduce 操作。这比起过程式的语言来说，在代码上要更容易阅读。（传统过程式的语言需要使用 for/while 循环，然后在各种变量中把数据倒过来倒过去的）这个很像 C++ STL 中 foreach、find_if、count_if 等函数的玩法。

- **pipeline（管道）**：这个技术的意思是，将函数实例成一个一个的 action，然后将一组 action 放到一个数组或是列表中，再把数据传给这个 action list，数据就像一个 pipeline 一样顺序地被各个函数所操作，最终得到我们想要的结果。

- **recursing（递归）**：递归最大的好处就简化代码，它可以把一个复杂的问题用很简单的代码描述出来。注意：递归的精髓是描述问题，而这正是函数式编程的精髓。

- **currying（柯里化）**：将一个函数的多个参数分解成多个函数，然后将函数多层封装起来，每层函数都返回一个函数去接收下一个参数，这可以简化函数的多个参数。在 C++ 中，这很像 STL 中的 bind1st 或是 bind2nd。

- **higher order function（高阶函数）**：所谓高阶函数就是函数当参数，把传入的函数做一个封装，然后返回这个封装函数。现象上就是函数传进传出，就像面向对象对象满天飞一样。这个技术用来做 Decorator 很不错。

上面这些技术太抽象了，我们还是从一个最简单的例子开始。

```
// 非函数式，不是 pure funciton，有状态
int cnt;
void increment(){
    cnt++;
}
```

这里有个全局变量，调这个全局函数变量 ++，这里面是有状态的，这个状态在外部。所以，如果是多线程的话，这里面的代码是不安全的。

如果写成纯函数，应该是下面这个样子。

```
// 函数式，pure function， 无状态
int increment(int cnt){
    return cnt+1;
}
```

这个是你传给我什么，我就返回这个值的 +1 值，你会发现，代码随便拷，而且与线程无关，代码在并行时候不用锁，因为是复制了原有的数据，并返回了新的数据。

我们再来看另一个例子：

```
def inc(x):
    def incx(y):
        return x+y
    return incx

inc2 = inc(2)
inc5 = inc(5)

print inc2(5) # 输出 7
print inc5(5) # 输出 10
```

上面这段 Python 的代码，开始有点复杂了。我们可以看到上面那个例子`inc()`函数返回了另一个函数`incx()`，于是可以用`inc()`函数来构造各种版本的 inc 函数，比如：`inc2()`和`inc5()`。这个技术其实就是上面所说的 currying 技术。从这个技术上，你可能体会到函数式编程的理念。

- **把函数当成变量来用，关注描述问题而不是怎么实现，这样可以让代码更易读。**

- **因为函数返回里面的这个函数，所以函数关注的是表达式，关注的是描述这个问题，而不是怎么实现这个事情。**

Lisp 语言介绍

要说函数式语言，不可避免地要说一下 Lisp。

下面，我们再来看看 Scheme 语言（Lisp 的一个方言）的函数式玩法。在 Scheme 里，所有的操作都是函数，包括加减乘除这样的东西。所以，一个表达式是这样的形式——（函数名 参数 1 参数 1）

```
(define (plus x y)  (+ x y))
(define (times x y) (* x y))
(define (square x)  (times x x))
```

上面三个函数：

- 用内置的 + 函数定义了一个新的 `plus` 函数。
- 用内置的 * 函数定义了一个新的 `times` 函数。
- 用之前的 `times` 函数定义了一个 `square` 函数。

下面这个函数定义了：f(x) = 5 * x^2 +10

```
(define (f1 x) ;;; f(x) = 5 * x^2 + 10
    (plus 10 (times 5 (square x))))
```

也可以这样定义——使用 lambda 匿名函数。

```
(define f2
    (lambda (x)
        (define plus
            (lambda (a b) (+ a b)))
        (define times
            (lambda (a b) (* a b)))
        (plus 10 (times 5 (times x x)))))
```

在上面的这个代码里，我们使用 lambda 来定义函数 `f2`，然后也同样用 lambda 定义了两个函数——`plus` 和 `times`。最后，由 `(plus 10 (times 5 (times x x)))` 定义了 `f2`。

我们再来看一个阶乘的示例：

```
;;; recursion
(define factoral (lambda (x)
    (if (<= x 1) 1
        (* x (factoral (- x 1))))))

(newline)
(display(factoral 6))
```

下面是另一个版本的，使用了尾递归。

```
;;; another version of recursion
(define (factoral_x n)
    (define (iter product counter)
        (if (< counter n)
            product
            (iter (* counter product) (+ counter 1))))
    (iter 1 1))

(newline)
(display(factoral_x 5))
```