

The Why of Y

Richard P. Gabriel

Lucid, Inc. and Stanford University

翻译：硅胶鱼

你有没有好奇过 Y 是如何工作的以及怎么会有人能把牠给搞出来？在这篇文章里我会试着向你解释牠是如何工作的，以及更牛的——神牛是怎么把牠发明出来的，我将使用 Scheme 语言的写法，因为把函数作为参数传递的时候，这样写更容易理解。

Y 的意义就是提供了不使用内建方法实现自引用程序的机制（就是说 Y 能实现函数的匿名递归——译者注）。Scheme 语言提供了几种实现递归的机制，比如定义全局函数和使用关键字 **letrec**。下面就是 Scheme 中阶乘函数的一种写法：

```
(define fact
  (lambda (n)
    (if (< n 2) 1 (* n (fact (- n 1))))))
```

这段程序能正确运行是因为全局变量 **fact** 的值被传递给了 lambda 表达式。当计算函数体内的变量 **fact** 以决定调用哪个函数的时候，**fact** 的值在全局变量中被找到。从某种意义上来说，使用全局变量是让人很不爽的，因为这要依赖一个全局的，因而脆弱的资源——全局变量空间(global variable space)。

在 Scheme 中，用关键字 **letrec** 实现自引用通常得依靠副作用(side effect)。没有副作用的编程语言和程序是更容易理解的。因此，学会写不依赖副作用的递归程序是有一定的理论意义的。

The following is a program that uses **letrec**:

下面是个使用关键字 **letrec** 的程序：

```
(letrec
  ((f (lambda (n)
        (if (< n 2) 1 (* n (f (- n 1)))))))
  (f 10))
```

这个程序是用来计算 10! 的。Lambda 表达式中的 **f** 引用的是 **f** 的绑定（the binding of **f**，就是那个 lambda 表达式——译者注）。这个引用关系是靠关键字 **letrec** 建立起来的。

我们可以用关键字 `let` 和 `set!` 来实现关键字 `letrec`。

```
(letrec ((f (lambda ...))) ...)
```

这和下面的程序的等价的：

```
(let ((f <undefined>)) (set! f (lambda ...)) ...)
```

Lambda 表达式中，所有对 `f` 的引用都将会引用那个 lambda 表达式的值。

`Y` 是个函数，牠接收一个函数作为参数，这个函数可以看作是对需要实现的递归或自引用的描述。`Y` 返回另一个实现了这个递归的函数。下面是 `Y` 如何被用于计算 `10!`。

```
(let ((f (y (lambda (h)
              (lambda (n)
                (if (< n 2) 1 (* n (h (- n 1))))))))
    (f 10))
```

注意那个作为参数传递给 `Y` 的函数：这个函数接受一个函数作为参数，然后返回一个看起来很像是我们想要定义的阶乘函数的函数。也就是说传递给 `Y` 的函数是那个 `(lambda (h) ...)`，这个函数的函数体看起来像阶乘函数，区别只是它在阶乘函数应该递归调用阶乘函数的地方调用了 `h`。`Y` 为 `h` 安排一个适当的值。

`Y` 一般被称为函数的应用序不动点算子 (applicative-order fixed point operator for functionals)。现在来看看，在我们的阶乘函数中，这意味着什么。假设 `f` 是数学意义上正确的阶乘函数，或许存在于柏拉图的理想空间中。让 `F` 表示以下函数：

```
F = (lambda (h) (lambda (n) (if (< n 2) 1 (* n (h (- n 1))))))
```

可得 $((F\ f)\ n) = (f\ n)$ 。也就是说，`f` 是 `F` 的一个不动点：`F` 是(从某种意义上来说) `f` 到 `f` 的自身映射。`Y` 满足下面这个性质： $((F\ (Y\ F))\ x) = ((Y\ F)\ x)$ 。这是 `Y` 非常重要的一个性质。另一个重要性质是函数的最小定义的不动点是惟一的，所以 `(Y F)` 和 `f` 在某种意义上是一样的。

应用序的 `Y` 和经典的 `Y` 组合子是不一样的。在一些教材里，我们这里所说的 `Y` 被称为 `Z`。

为了推导出 `Y`，我将从一个递归函数的具体例子开始：阶乘函数。在推导过程中，我将使用三种技术。首先是通过传递一个额外的参数来避免使用 Scheme 语言本身提供的机

制实现自引用。第二项技术是将有多个参数的函数转化成嵌套的单参数函数(也就是传说中的 currying——译者注),从而分离对自引用参数(第一项技术引入的参数——译者注)的操作和对普通参数的操作。第三是把函数抽象出来。

下面所有代码都使用 n 和 m 代表整数, 变量 x 是个未知的无类型参数, 变量 f, g, h, q 以及 r 代表函数。

下面是阶乘函数的基本形式:

```
(lambda (n) (if (< n 2) 1 (* n (h (- n 1)))))
```

变量 h 表示当递归调用发生时我们想要调用的那个函数, 即阶乘函数本身。由于我们无法让 h 直接调用阶乘函数, 所以把它作为参数传递进去:

```
(lambda (h n) (if (< n 2) 1 (* n (h h (- n 1)))))
```

在对 h 的递归调用中, 它的第一个参数也是 h 。这是因为随后的函数调用中, 我们要传递在递归过程中要使用的恰当的函数。

因此, 为了计算 $10!$, 我们写出代码如下:

```
(let ((g (lambda (h n)
            (if (< n 2) 1 (* n (h h (- n 1)))))
      (g g 10)))
```

在对 g 的函数体求值时, h 的值和 `let` 关键字确立的 g 的值是一样的;也就是说, 在执行 g 的时候, h 指的是正在执行中的函数。当函数调用 $(h h (- n 1))$ 发生时, 这个相同的值被作为参数直接传递给了 h : h 把它自己传递给了自己。

可是, 我们想要做的是把函数自引用的操作和对其它普通参数的操作分离开来。这里, 我们希望把对 h 的操作和对 n 的操作分离开。解决这个问题的一般方案是使用一种被称为 *科里化*(currying)的技术。在把这个例子科里化之前, 我们先来看看另一个使用科里化技术的例子。下面的程序用看起来更聪明一些的方法计算 $10!$ 。

```
(letrec ((f (lambda (n m)
              (if (< n 2) m (f (- n 1) (* m n)))))
  (f 10 1))
```

这里使用了一个小技巧：用累积参数 *m* 来计算结果。这个函数在 Scheme 中是迭代计算的，（而不是递归计算的）这点并不重要。现在把 *f* 的定义科里化：

```
(letrec ((f (lambda (n)
              (lambda (m)
                (if (< n 2) m ((f (- n 1)) (* m n)))))))
  ((f 10) 1))
```

科里化的主要思想就是让每个函数都只有一个参数，传递多个参数则通过嵌套函数的计算实现：第一次计算返回一个接受第二个参数的函数并完成求值。在上面的那段代码中递归调用 `((f (- n 1)) (* m n))` 有两个步骤：先计算出满足要求的函数，然后将它应用于参数。

我们可以用这个思想来科里化阶乘函数：

```
(let ((g (lambda (h)
            (lambda (n)
              (if (< n 2) 1 (* n ((h h) (- n 1)))))))
  ((g g) 10))
```

在这段代码中，递归调用有两个步骤，第一步是计算出满足我们要求的函数。但是，这个函数是通过把一个函数应用于其自身计算出来的。

我们通过把一个函数应用于其自身来达成自引用的基本功能。程序最后一行中的 `(g g)` 中，*g* 把它自己作为参数。它返回一个闭包，闭包中的变量 *h* 受限於外面的 *g*。这个闭包接受一个数字，然后进行基本的阶乘运算。如果这个运算需要使用递归调用，它会调用把 *h* 当成参数的 *h*，但是这些 *h* 都受限於 `let` 关键字定义的函数 *g*。

我们来总结一下这个技巧。假设有个像下面的代码框架一样使用关键字 `letrec` 的自引用函数：

```
(letrec ((f (lambda (x) ... f ...))) ... f ...)
```

那么它可以改写成使用关键字 `let` 的自引用函数：

```
(let ((f (lambda (r) (lambda (x) ... (r r) ...)))) ... (f f) ...)
```

r 是一个新变量。

让我们集中精力看看，怎么进一步分离阶乘函数中对 h 的操作和对 n 的操作。回忆一下，阶乘函数是这样的：

```
(let ((g (lambda (h)
  (lambda (n)
    (if (< n 2) 1 (* n ((h h) (- n 1)))))))
  ((g g) 10))
```

我们的攻坚计划是把 if 表达式从 $(h\ h)$ 和 n 抽象出来。这可以达到两个目的：我们得到的函数将会独立于外面的绑定，对控制参数和数字参数的操作互相分离。抽象出来的结果如下：

```
(let ((g (lambda (h)
  (lambda (n)
    (let ((f (lambda (q n)
      (if (< n 2) 1 (* n (q (- n 1)))))
      (f (h h) n))))))
  ((g g) 10))
```

我们把 f 的定义科里化，这也将会改变对它的调用。

```
(let ((g (lambda (h)
  (lambda (n)
    (let ((f (lambda (q)
      (lambda (n)
        (if (< n 2) 1 (* n (q (- n 1)))))
      ((f (h h)) n))))))
  ((g g) 10))
```

可以看到，这里 f 的定义并不用像这样深深地嵌套在函数 g 里。因此，我们把函数的主要部分（计算阶乘的那部分）抽取出来。

```
(let ((f (lambda (q) (lambda (n) (if (< n 2) 1 (* n (q (- n 1)))))
  (let ((g (lambda (h) (lambda (n) ((f (h h)) n))))
    ((g g) 10)))
```

注意以下两点：首先， f 的形式又一次变成了阶乘函数的参数化形式；其次，我们可以把这个表达式从 f 抽象出来，这就产生了 Y ：

```
(define Y (lambda (f)
  (let ((g (lambda (h)
    (lambda (x) ((f (h h)) x)))))
    (g g))))
```

以上就是推导出 Y 的一种方法。