[读书笔记]SICP：9[B]赋值和局部状态

本章对应于书中的3.1。

函数式程序与命令式程序的区别：

- **函数式程序：**不使用赋值语句，能使用代换模型这一计算模型，当使用相同参数对统一过程进行调用时，会产生相同的结果，可以认为过程是在计算数学函数。
- **命令式程序：**使用赋值语句
- - **好处：**只有具有赋值操作`set!`，才能对自己的局部状态变量进行更新赋值，且将其封装起来，避免需要更新局部状态变量时需要将其作为参数进行迭代更新，使得将自己随时间变化的内部状态隐藏起来，由此使得实现细节完全独立于程序的其他部分，可以以一种更模块化的方法构建系统。
  - **缺点：**由于引入了赋值操作，则变量的值就可以改变，一个变量就不再是一个简单的名字，而是索引着一个可以保存值的位置，而存储在那里的值也可以改变，就无法使用代换模型这一计算模型。并且引入赋值操作，会破坏语言的引用透明性，就无法确定“同一”这一概念。并且引入赋值操作后，需要小心赋值操作的顺序，否则会出现函数式程序不会出现的错误，这同样会使得程序设计复杂化，使其中的主要思想变模糊。

这里还需要一些组织原则来指导我们系统化地完成系统的整体设计。我们可以基于被模拟系统的结构去设计程序的结构，对于有关的物理系统中的每个对象，我们构造对应的计算对象；对该系统中的每种活动，就定义之中符号操作。这里有两种不同的世界观：

- 关注对象，将大型系统看成一大批对象，它们的行为可能随时间的进展不断变化。需要关注计算对象可以怎样变化有同事保持其标识，迫使我们抛弃代换模型，转向环境模型。
- 关注流过系统的信息流，能够松弛模型中对时间的模拟与计算求职过程中的各种事件发生的顺序。

这一章主要关注赋值和局部状态，这里我们将世界看成是许多独立对象的集合，每个对象都有自己随时间变化的状态，我们可以用若干个**状态变量**来刻画一个对象的状态，由此能够确定该对象当前的行为。而在一个由多个对象组成的系统中，对象之间通常会通过**交互作用**来影响其他对象的状态，这里的交互就是建立一个对象的状态变量与其他对象的状态变量之间的联系。

我们可以根据这种观点来组织该系统计算模型的框架，想要模型具有模块化，则要将其分解为一系列计算对象来模拟系统中的实际对象，每个计算对象必须有自己的一些**局部状态变量**，来描述实际对象的状态。并且由于实际对象的状态是随时间变化的，所以对应的计算对象的状态也要随时间变化。此外，如果我们希望通过程序设计语言里的符号来模拟状态变量，则该语言需要提供一个**赋值运算符**，由此来改变与一个名字相关的值。

1 局部状态变量

这里以模拟从银行取现金为例，通过传递`amount`参数给`withdraw`过程，来模拟从银行中取`amout`现金，并且这里还需要一个变量`balance`来表示账户中的现金。

注意：`withdraw`执行结果与`balance`的值有关，无法保证每次执行都得到相同的结果。

对应的代码如下：

```
(define balance 100)
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds"))
```

- `set!`：这个为**赋值操作**，`为符号，`为表达式，`set!`修改`将其值变为`求值的结果。
- `begin ...`：按需求值，并将最后的`求值结果作为`begin`的求值结果。

通过这种形式的代码可以模拟从银行取现金的过程，但是由于`balance`为全局变量，可以被任意的修改，我们希望`balance`是只能由`withdraw`使用的局部状态变量，用来保存账户状态变化的轨迹。

- **理想**：我们希望只有当前过程能修改局部状态变量，且该局部状态变量能随时间变化
- **方法**：所以需要将该局部状态变量定义在过程内部（即在过程内部对该局部状态变量进行初始化），但是每次调用该过程就会对局部状态变量进行重新初始化，使得局部状态变量无法随时间变化，因此需要将局部状态变量定义在该过程的外部，但是为了使得只有当前过程能修改该局部状态变量，可以在该过程外面包裹一个过程，然后在外部过程内、内部过程外定义局部状态变量，由此只有第一次调用外部过程会初始化局部状态变量，而后只有调用内部过程才会修改局部状态变量，且不会反复初始化该局部状态变量，且直接使用外部过程就能得到随时间变化到当前的局部状态变量，使得该局部状态变量能随时间变化。即`外部过程（初始化局部状态变量）、内部过程（修改局部状态变量）`。
- **实现**：可以通过在外部过程中使用`let`、`define`、形参来定义局部状态变量。

这里使用`let`来定义局部状态变量，新的`withdraw`过程为

```
(define new-withdraw  ; 注意不能有括号
 (let ((balance 100))
   (lambda (amount)
     (if (>= balance amount)
         (begin (set! balance (- balance amount))
                balance)
         "Insufficient funds"))))
```

这里的外部过程是`new-withdraw`，内部过程通过`lambda`实现，而`let`在`new-withdraw`和`lambda`定义了局部状态变量`balance`。

注意：这里无法用代换模型进行解释，后面会介绍新的方法。

我们同样可以用形参来定义局部状态变量

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))
```

此时就能传递`balance`到`make-withdraw`过程，来得到不同的局部状态变量。如

```
(define W1 (make-withdraw 100))
(define W2 (make-withdraw 200))
```

则`W1`和`W2`就维护了自己独立的状态变量。

同样也可以定义一个银行账户

```
(define (make-account balance secret-password)
  (define count 0)
  (define (call-the-cops) "Call the cops")
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
```

```
    balance)

  (define (dispatch password m)
    (cond ((not (eq? password secret-password))
           (lambda (x)
             (if (>= count 2)
               (call-the-cops)
               (begin (set! count (+ count 1))
                      "Incorrect password"))))
          ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request: MAKE-ACCOUNT" m))))
  dispatch)
```

这里定义了两个局部状态变量`balance`和`secret-password`。

由此就将系统看成是一群带有局部状态的对象，提供了一种维护模块化设计的强有力技术。

2 赋值带来的好处

我们这里以蒙特卡罗模拟为例来说明赋值带来的好处。有个定理表示：随机两个整数没有公共因子的概率为$6/\pi^2$。则我们可以不断随机产生两个随机数，然后判断该随机数是否含有公共因子，实验多次后近似于$6/\pi^2$，由此就能近似于π。

首先，我们按照以上的方式，我们定义一个产生随机数的`rand`过程，它需要一个局部状态变量来表示当前的随机数，然后通过`rand-update`过程来得到下一个随机数，这里可以通过`let`来初始化定义局部变量，然后通过`set!`进行赋值操作，则该随机数生成器为

```
(define rand
  (let ((x rand-init))
    (lambda ()
      (set! x (rand-update x))
      x)))
```

则对应的蒙地卡罗模拟代码为

```
(define (estimate-pi trials)
  (sqrt (/ 6 (monte-carlo trials cesaro-test))))
(define (cesaro-test)
   (= (gcd (rand) (rand)) 1))

(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0) (/ trials-passed trials))
          ((experiment)
           (iter (- trials-remaining 1)
                 (+ trials-passed 1)))
          (else
           (iter (- trials-remaining 1)
                 trials-passed))))
  (iter trials 0))
```

如果我们这里不采用赋值操作`set!`，则无法在`rand`中不断修改当前的局部状态变量`x`，就无法计算出下一时刻的局部状态变量`x`，所以此时就需要显示地计算下一时刻的局部状态变量，并将其作为参数，传入下一次迭代中，才能让局部状态变量随时间更新，此时需要显示进行维护。 对应的代码如下

```
(define (estimate-pi trials)
```

```
  (sqrt (/ 6 (random-gcd-test trials random-init))))

(define (random-gcd-test trials initial-x)
  (define (iter trials-remaining trials-passed x)
    (let ((x1 (rand-update x)))
      (let ((x2 (rand-update x1)))
        (cond ((= trials-remaining 0) (/ trials-passed trials))
              ((= (gcd x1 x2) 1)
               (iter (- trials-remaining 1)
                     (+ trials-passed 1)
                     x2))
              (else
               (iter (- trials-remaining 1)
                     trials-passed
                     x2))))))
  (iter trials 0 initial-x))
```

这里由于没有使用赋值操作，所以每次需要显示地计算`x1`和`x2`，然后将`x2`作为当前的局部状态变量传递到下一次迭代中。这就使得对随机数的生成处理与我们的业务代码交织在一起，使得能难将蒙地卡罗方法的思想独立出来，而之前使用赋值操作的代码，它通过赋值操作能将局部状态变量的更新隔离在`rand`中，无需每次显示传递下一时刻的局部状态变量。

赋值操作的好处：只有具有赋值操作`set!`，才能对自己的局部状态变量进行更新赋值，且将其封装起来，避免需要更新局部状态变量时需要将其作为参数进行迭代更新，使得将自己随时间变化的内部状态隐藏起来，由此使得实现细节完全独立于程序的其他部分，可以以一种更模块化的方法构建系统。

3 引入赋值操作的代价

我们这里以两个代码为例，其中一个是有赋值操作，另一个不具有赋值操作

```
; 有赋值操作
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
(define W (make-simplified-withdraw 25))
(W 20) ; 5
(W 10) ; -5
; 无赋值操作
(define (make-decrementer balance)
  (lambda (amount) (- balance amount)))
(define D (make-decrementer 25))
(D 20) ; 5
(D 10) ; 15
```

这种不使用赋值操作的程序设计称为**函数式程序设计**，当使用相同参数对统一过程进行调用时，会产生相同的结果，可以认为过程是在计算数学函数。当我们对两者都使用代换模型进行解释时，过程如下

```
((make-decrementer 25) 20)
; ((lambda (amount) (- 25 amount)) 20) ; 用25代换balance
; (- 25 20) ; 用20代换amount
; 5
((make-simplified-withdraw 25) 20)
; ((lambda (amount) (set! balance (- 25 amount)) 25) 20) ; 用25代换balance， 这里不代换
set!中的balance，否则没有意义
; (set! balance (- 25 20)) 25
; (set! balance 5) 25 ; 显示25， 而将5赋值给balance
```

可以发现没有使用赋值操作时通过代换模型能进行解释，而使用了赋值操作就无法使用代换模型进行解释。**根本原因在于**：代换模型要求语言中的符号只是作为值的名字，如果使用了赋值操作，则变量的值就可以改变，一个变量就不再是一个简单的名字，而是索引着一个可以保存值的位置，而存储在那里的值也可以改变。这就要求必须区分`set!`作用前和`set!`作用后的变量`balance`，而代换模型做不到。

考虑两个物体“同一”（the same）的概念。如果一个语言支持表达式里“同一对象可以互相替换”且不会改变有关表达式的值，则该语言具有**引用透明性**，当计算机语言中包含`set!`后，确定对象是否“同一”就变得很难定义，且看上去“同一的”对象也无法互相替换而不改变表达式的值，这就打破了引用透明性，此时能难通过等价的表达式代换去简化表达式，也就打破了代换模型这种计算模型。以下面4个对象为例

```
; 无赋值过程
(define D1 (make-decrementer 25))
(define D2 (make-decrementer 25))
; 有赋值过程
(define W1 (make-simplified-withdraw 25))
(define W2 (make-simplified-withdraw 25))
```

没有赋值过程的`D1`和`D2`没有打破引用透明性，两者由于具有相同的计算行为所以是“同一的”，可以在任意表达式中将两者互换而不影响表达式的结果。而使用了赋值过程的`W1`和`W2`虽然是通过相同表达式的求值创建起来的，但无法在任意表达式中互换而不改变表达式的结果，由此打破了引用透明性，由此无法将两者互换而不影响表达式的结果。当不具有引用透明性时，就很难确定对象是否“同一的”：

- 想要判断对象是否“同一的”，可以改变其中一个对象，看另一个对象是否改变
- 想要看一个对象是否改变，需要观察“同一”对象两次，才能确定
- 想要观察“同一”对象，就需要先判断该对象是否“同一的”，又回到第一个问题了

通常当我们不改变数据对象时，可以将一个复合数据对象完全看成是由它的各个组成部分构成的整体，比如一个有理数可以由它的分子和分母完全确定。如果修改数据对象时，该观点就不成立了，此时复合数据对象不是由它的各个组成部分确定的，它本身具有一个“标志”。比如以下代码

```
; 情况一
(define peter-acc (make-account 100))
(define paul-acc (make-account 100))
;情况二
(define peter-acc (make-account 100))
(define paul-acc peter-acc)
```

情况一定义了两个不同的银行账户，即使它们可能具有相同的`balance`，但是它们本身的“标志”不同，与它们的`balance`无关，两个银行账户做交易互不影响；而情况二定义了一个具有两个名字的银行账户，即使账户的`balance`改变了，它的“标志”也没有改变，仍然是“同一”对象。这种复杂性是由于将其作为对象导致的，它会通过赋值来修改它的局部状态变量。

这种广泛采用赋值的程序称为**命令式程序**设计，除了导致计算模型的复杂性外，还容易出现一些函数式程序不存在的错误，比如以下函数式程序可以改为对应的命令式程序

```
; 函数式程序
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
; 命令式程序
(define (factorial n)
  (let ((product 1)
        (counter 1))
    (define (iter)
```

```
    (if (> counter n)
        product
        (begin (set! product (* counter product))
               (set! counter (+ counter 1))
               (iter))))
  (iter)))
```

这里通过将`product`和`counter`当做局部状态变量，然后使用赋值语句来将其转化为命令式程序。但是要十分注意两个赋值语句的顺序，否则会出错。