3. 模块化，对象和状态(I)

本节课的这部分讨论：

- 赋值和局部状态
- 基于状态改变的程序设计
- 引入赋值的得与失
- 函数式和命令式程序设计
- 引入赋值的得与失
- 命令式程序设计的缺陷

对象和流：有用的设计策略

- 前面讨论了如何组合基本过程和基本数据，构造各种复合对象（过程/数据）。以及抽象在控制和处理程序复杂性中的重要作用
- 大型系统的有效设计还需要一些组织原则，以系统化地完成设计。特别是需要一些模块化策略把复杂系统划分为内聚力强的部分，使之可以分别开发和维护
- 一种适合构造模拟真实世界的系统的策略：根据被模拟系统设计程序结构
  - 针对实际物理系统中的每个对象，构造一个与之对应的程序对象
  - 针对实际系统里的每种活动，在计算系统里实现一种对应操作
- 采用这种策略时有一个重要设计目标：真实系统里增加新对象或操作时，无须大范围修改程序，只需局部修改，简单加入对象或操作
- 本章讨论两种组织策略：
  - 把系统看成由一批相互作用的对象组成，它们随着时间进展不断变化
  - 把系统看作一种信号处理系统，关注流过系统的信息流，
- 基于对象和基于流的设计途径都对语言提出了新要求
  - 基于对象要求实现在存在期间能变化但保持其标识的对象，是新计算模型
  - 基于流的技术要求一种延时求值技术

对象：状态和变化

- 对世界的一种看法：世界由一批事物(对象)组成，每个对象有其状态，有其行为方式。其状态随时间不断变化，其行为受到历史的影响
- 例：一个银行账户有状态，对“能取出100元吗”的回答依赖于该账户此前存钱和取钱的历史
- 模拟这种对象，程序里要用状态变量刻画计算对象的状态。选择状态变量（如，记录余额还是记录全部交易历史）的根据是对于对象行为的预期。状态变量应足够确定对象的后续行为
- 系统里的不同对象相互联系，通过交互影响彼此的状态。有些对象联系更紧密，可能形成分组，或构成大系统里的子系统
- 这种观点是组织系统的计算模型（程序）的一种有力手段。它倡导一种将系统模块化的方式：把一个系统分解为一组计算对象，用它们模拟真实世界中真实对象的行为，从而模拟真实系统的行为
- 真实对象随着时间而改变状态，计算对象要模拟它们。程序里就需要有随着运行不断改变状态的对象，需要改变程序对象状态的操作：赋值操作。赋值就是修改对象的状态变量

局部状态变量

- 考虑模拟一个银行账户：假定过程 withdraw 表达的是从该账户提取现金的操作。可能出现下面操作序列：

  (withdraw 25)
  75
  (withdraw 60)
  15
  (withdraw 25)
  "Insufficient funds"
  (withdraw 10)
  5

- 两次调用 (withdraw 25) 取 25 元，却得到截然不同的结果。说明这个过程与前面计算数学函数的过程的性质完全不同
- withdraw 的行为与时间有关，依赖于某个（某些）随时间而改变状态的变量。前面的操作历史影响后面操作的行为

变量和赋值

- 用变量 balance 表示账户余额，将 withdraw 定义为依赖它的过程：

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

  begin 是特殊形式，它逐个求值其参数，得到最后一个表达式的值

  (set! balance (- balance amount)) 完成修改 balance 的工作

  set! 是赋值运算符，一般形式是 (set! <*name*> <*new-value*>)

- 这里不能用 (define balance ...)

  set! 找到最近的已有定义的变量，修改它的值约束

  define 在其所在的定义域里创建变量的约束

  更具体的语义细节后面介绍

一点说明

- 后面多处用到一般形式的 lambda 表达式和 define 形式，它们的体起一个 begin 表达式的作用
- lambda 表达式的参数表之后允许写多个表达式，其一般形式是

  (lambda (x …) <exp1> … <expn>)

  执行时顺序求值表达式 <expi>，以最后表达式的值作为过程的值

- 相应用 define 定义过程的形式也类似：

  (define (f x …) <exp1> … <expn>)

  过程调用时逐个求值过程体中的表达式，以最后表达式的值为值

- 如果表达式不改变变量状态，允许写多个表达式完全没有必要
- 有了 set!，前面表达式的求值有可能影响后面的表达式，在过程体中写一系列的表达式就有用了。下面有许多这样的例子

局部状态变量

- 虽然前面定义可行，但也有些不妥： balance 作为全局名字，任何过程都能访问和修改它
- 可以把它改为 withdraw 里局部的东西：

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

  let 创建一个包含局部变量 balance 的环境，并将它初始化为100。new-withdraw 的功能和前面的 withdraw 一样

- set! 和局部变量结合形成了一种通用编程技术，下面将一直用这种技术构造带有局部状态的计算对象。这一技术也带来一个问题：代换模型对这种程序失效了，需要新的计算模型（后面介绍）
- 下面考虑 new-withdraw 的一些问题和变形

局部状态变量

- 把 new-withdraw 修改为一种创建“提款处理器”的过程：

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

  使用实例:

```
(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))
(W1 50)
50
(W2 70)
30
(W2 40)
"Insufficient funds"
(W1 40)
10
```

  注意：形参也是局部变量，定义 make-withdraw 时可以不用 let

局部状态变量

- 可扩充为创建银行账户的过程，使账户

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balan
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amou
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withd
          ((eq? m 'deposit) deposi
          (else (error "Unknown re
  dispatch)
```

使用实例：

```
(define acc (make-account 100))
((acc 'withdraw) 50)
50
((acc 'withdraw) 60)
"Insufficient funds"
((acc 'deposit) 40)
90
((acc 'withdraw) 60)
30
```

可创建任意多个独立的账户对象：

```
(define acc2 (make-account 200))
((acc2 'withdraw) 50)
150
```

这个过程返回一个带局部环境的对象（过程），以相应消息作为输入，该对象将返回过程 withdraw 或 deposit

赋值：得与失

- 赋值给程序的理解带来新问题。另一方面，把系统看成一组有内部状态的对象，也是实现模块化设计的强有力技术。下面是一个例子
- 实例：设计随机数生成过程 rand，希望反复调用能生成一系列整数，这些数具有均匀分布的统计性质
- 假定已有过程 rand-update，对一个数调用它，将得到另一个数

  x_2 = (rand-update x_1)
  x_3 = (rand-update x_2)

  反复做可以得到所需的整数序列
- rand 实现为一个带有局部状态变量的过程：

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

rand-init 应为某个整数（变量）

赋值：得与失

- 直接用函数 rand-update 也可以生成随机数，但用起来很麻烦
- 对比实例：用随机数实现蒙特卡罗模拟。
  - 蒙特卡罗方法：用大量随机数做试验，统计试验结果得到结论
  - 下面的具体试验：两个整数之间无公因子的概率是 $6/\pi^2$
- 考虑下面过程，核心过程 monte-carlo 的结构很清楚：

```
(define (estimate-pi trials)  ; trials: 试验次数
  (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))
```


实例：蒙特卡罗模拟

- 不用赋值而直接用 rand-update，也可以写出程序：

```
(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))
```

能行。只用函数式过程时，random-gcd-test 必须显式操作随机数 x1 和 x2，迭代时把 x2 作为新输入

蒙特卡罗模拟：分析

- 完成这个试验用到两个随机数，有的试验可能用三个或更多随机数，函数式写法必须时时注意维护这些随机数
- 代码里没有体现蒙特卡罗方法本身（无此概念），它与其他操作交织在一起。在用 rand 的版本里蒙特卡罗方法是独立过程，随机数的使用细节屏蔽在这个过程内部
- 另一个现象：在复杂计算中，从一个部分的角度看，其他部分都像是在随时间不断变化。它们通常隐藏起其变化细节（内部状态）
- 要用这种方式分解系统，最直接方式就是用计算对象模拟系统随时间变化的行为，用局部变量模拟子部分的内部状态，用赋值模拟状态变化
- 对这个例子的简单总结。讨论了处理状态变化问题有两种方式：
  - 通过显式计算和额外的参数传递随时间变化的状态
  - 通过局部状态变量和赋值

  后一方式很可能得到更模块化的系统。但也带来许多麻烦（见下）

引进赋值的代价

- 用了赋值操作 set! 后，编程语言就不能用简单的代换模型解释了，而且，任何具有漂亮的数学性质的简单模型，都不可能作为描述这种程序里的对象和赋值的理论框架
- 没有赋值时，以同样参数调用同一过程总得到同样结果。这样的过程就像在计算数学的函数。无赋值的编程称为函数式编程
- 看下面过程（计算简单的数学函数）：

```
(define (make-decrementer balance)
  (lambda (amount)
    (- balance amount)))

(define D (make-decrementer 25))
(D 20)
5
(D 10)
15
(D 10)
15
```


引进赋值的代价

- 与下面精简版的 withdraw 过程比较

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
(define W (make-simplified-withdraw 25))
(W 20)
5
(W 10)
-5
(W 10)
-15
```

- 代换模型可以解释 make-decrementer，无法解释 make-simplified-withdraw，不能解释为什么两个(W 10) 调用 会得到不同结果
- 在代换模型里名字是值的代号，可以用代换全部消除掉。有赋值后变量就不是代表值的简单名字，而是保存值的位置，其值可以改变

同一和变化

- 赋值打破了简单的计算模型，但其意义还更深远。计算模型里引入了变化之后，许多基本概念都出了问题
- 第一个问题：什么是同一个（sameness）？
- 用同样参数两次调用 make-decrementer，得到的是同一个东西吗？

```
(define D1 (make-decrementer 25))
(define D2 (make-decrementer 25))
```

- 虽然 D1 和 D2 名字不同，但它们永远表现出同样行为，在任何计算中都可以任意相互替代而不会导致可以察觉的不同
- 调用 make-simplified-withdraw 两次，得到：

```
(define W1 (make-simplified-withdraw 25))
(define W2 (make-simplified-withdraw 25))
```

- W1 和 W2 有独立行为，在程序里不能任意相互替代

例：

```
(W1 20)
5
(W1 20)
-15
(W2 20)
5
```

同一和变化

- 如果某种语言支持“同样的东西可以相互替换”，这种替换绝不会改变表达式的值，则称这种语言具有引用透明性
- 赋值打破了引用透明性，使确定一个替换会不会改变表达式的意义变得很困难，对程序进行推理也很困难
- 抛弃了引用透明性，“同一个”的概念也变得很复杂了。现实生活中也是如此，弄清什么是同一个事物也是很困难的
- 下面实例说明这一问题对于编程的影响。假定 Paul 和 Peter 有银行账户，其中有 100 块钱。下面是这一事实的两种模拟：

```
(define peter-acc (make-account 100))
(define paul-acc (make-account 100))
```

```
(define peter-acc (make-account 100))
(define paul-acc peter-acc)
```

- 两种情况下，Paul 和 Peter 都看到自己账号里有 100 元。但随后的提款活动却会让他们发现两种情况是不同的

同一和变化

- 构造计算模型时，这两种情况很容易搞混。特别是两个账号共享时，如果使用改变状态的操作（赋值），对一个账号的操作会影响另一个账号的行为，而这种影响没有在程序里明确写出
- 程序里两个不同描述实际指同一个东西时，称为别名（aliasing）。当不同数据结构之间出现共享时，从一条途径出发修改可能产生“修改”了另一数据结构的“副作用”。如果对这件事不清楚，就可能由于疏忽造成程序中的错误（副作用错误）
- 另一情况：如果 Paul 和 Peter 只能检查账户而不能取款（只读账户，其他人也不能改这两个账户），是否还应认为这两种模拟不同呢？
- 如前面的有理数对象，一旦建立后，其内容永远也不会变。这就是所谓“不变对象”（另如 Java 的字符串或 Integer 对象）
- 在这种情况下，究竟是实际上引用了两个内容相同的对象，还是正好引用到同一个对象，并没有实质性差别

命令式编程的缺陷

- 与函数式程序设计对应，广泛采用赋值的程序设计被称为命令式程序设计。命令式编程是常规软件开发中使用最广泛的编程范式
- 对命令式程序设计，除需要更复杂的计算模型来解释外，还很容易出现一些在函数式程序设计中不会出现的错误，即那种与操作的时间有关的错误（因为操作的效果依赖于历史）
- 重看前面的迭代式求阶乘过程：

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product) (+ counter 1))))
  (iter 1 1))
```

- 用命令式技术写，可以直接通过赋值修改 product 和 counter 的值，不必通过参数传递它们

命令式编程的缺陷

- 按命令式方式写出的程序是：

```
(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)))
```

- 这里有两个赋值，调换它们的顺序还能得到正确的程序吗？

```
(set! counter (+ counter 1))
(set! product (* counter product))
```

- 不行！命令式程序设计里赋值的顺序非常重要，而函数式程序设计不会在出现这种问题。后面还会看到命令式程序设计更多的问题