第二章　基础知识

这一节介绍一下 elisp 编程中一些最基本的概念，比如如何定义函数，程序的控制结构，变量的使用和作用域等等。

2.1　函数和变量

elisp 中定义一个函数是用这样的形式：

```
(defun function-name (arguments-list)
  "document string"
  body)
```

比如：

```
(defun hello-world (name)
  "Say hello to user whose name is NAME."
  (message "Hello, %s" name))
```

其中函数的文档字符串是可以省略的。但是建议为你的函数（除了最简单，不作为接口的）都加上文档字符串。这样将来别人使用你的扩展或者别人阅读你的代码或者自己进行维护都提供很大的方便。

在 emacs 里，当光标处于一个函数名上时，可以用 C-h f 查看这个函数的文档。比如前面这个函数，在 `*Help*` 缓冲区里的文档是：

```
hello-world is a Lisp function.
(hello-world name)

Say hello to user whose name is name.
```

如果你的函数是在文件中定义的。这个文档里还会给出一个链接能跳到定义的地方。

要运行一个函数，最一般的方式是：

```
(function-name arguments-list)
```

比如前面这个函数：

```
(hello-world "Emacser")                 ; => "Hello, Emacser"
```

每个函数都有一个返回值。这个返回值一般是函数定义里的最后一个表达式的值。

elisp 里的变量使用无需象 C 语言那样需要声明，你可以用 setq 直接对一个变量赋值。

```
(setq foo "I'm foo")                    ; => "I'm foo"
(message foo)                           ; => "I'm foo"
```

和函数一样，你可以用 C-h v 查看一个变量的文档。比如当光标在 foo 上时用 C-h v 时，文档是这样的：

```
foo's value is "I'm foo"


Documentation:
Not documented as a variable.
```

有一个特殊表达式（ special form）defvar，它可以声明一个变量，一般的形式是：

```
(defvar variable-name value
  "document string")
```

它与 setq 所不同的是，如果变量在声明之前，这个变量已经有一个值的话，用 defvar 声明的变量值不会改变成声明的那个值。另一个区别是 defvar 可以为变量提供文档字符串，当变量是在文件中定义的话，C-h v 后能给出变量定义的位置。比如：

```
(defvar foo "Did I have a value?"
  "A demo variable")                    ; => foo
foo                                     ; => "I'm foo"
(defvar bar "I'm bar"
  "A demo variable named \"bar\"")      ; => bar
bar                                     ; => "I'm bar"
```

用 C-h v 查看 foo 的文档，可以看到它已经变成：

```
foo's value is "I'm foo"


Documentation:
A demo variable
```

由于 elisp 中函数是全局的，变量也很容易成为全局变量（因为全局变量和局部变量的赋值都是使用 setq 函数），名字不互相冲突是很关键的。所以除了为你的函数和变量选择一个合适的前缀之外，用 C-h f 和 C-h v 查看一下函数名和变量名有没有已经被使用过是很关键的。

2.2 局部作用域的变量

如果没有局部作用域的变量，都使用全局变量，函数会相当难写。elisp 里可以用 let 和`let*`进行局部变量的绑定。let 使用的形式是：

```
(let (bindings)
  body)
```

bingdings 可以是 (var value) 这样对 var 赋初始值的形式，或者用 var 声明一个初始值为 nil 的变量。比如：

```
(defun circle-area (radix)
  (let ((pi 3.1415926)
        area)
```

```
    (setq area (* pi radix radix))
    (message "直径为 %.2f 的圆面积是 %.2f" radix area)))
(circle-area 3)
```

C-h v 查看 area 和 pi 应该没有这两个变量。

let* 和 let 的使用形式完全相同，唯一的区别是在let*声明中就能使用前面声明的变量，比如：

```
(defun circle-area (radix)
  (let* ((pi 3.1415926)
         (area (* pi radix radix)))
    (message "直径为 %.2f 的圆面积是 %.2f" radix area)))
```

2.3 lambda 表达式

可能你久闻 lambda 表达式的大名了。其实依我的理解，lambda 表达式相当于其它语言中的匿名函数。比如 perl 里的匿名函数。它的形式和 defun 是完全一样的：

```
(lambda (arguments-list)
  "documentation string"
  body)
```

调用 lambda 方法如下：

```
(funcall (lambda (name)
           (message "Hello, %s!" name)) "Emacser")
```

你也可以把 lambda 表达式赋值给一个变量，然后用 funcall 调用

```
(setq foo (lambda (name)
            (message "Hello, %s!" name)))
(funcall foo "Emacser")                    ; => "Hello, Emacser!"
```

lambda 表达式最常用的是作为参数传递给其它函数，比如 mapc。

2.4 控制结构

2.4.1 顺序执行

一般来说程序都是按表达式顺序依次执行的。这在 defun 等特殊环境中是自动进行的。但是一般情况下都不是这样的。比如你无法用 eval-last-sexp 同时执行两个表达式，在 if 表达式中的条件为真时执行的部分也只能运行一个表达式。这时就需要用 progn 这个特殊表达式。它的使用形式如下：

```
(progn A B C ...)
```

它的作用就是让表达式 A, B, C 顺序执行。比如：

```
(progn
  (setq foo 3)
  (message "Square of %d is %d" foo (* foo foo)))
```

2.4.2 条件判断

elisp 有两个最基本的条件判断表达式 if 和 cond。使用形式分别如下：

```
(if condition
    then
  else)

(cond (case1 do-when-case1)
      (case2 do-when-case2)
      ...
      (t do-when-none-meet))
```

使用的例子如下：

```
(defun my-max (a b)
  (if (> a b)
      a b))
(my-max 3 4)                          ; => 4

(defun fib (n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (t (+ (fib (- n 1))
              (fib (- n 2))))))
(fib 10)                              ; => 55
```

还有两个宏 when 和 unless，从它们的名字也就能知道它们是作什么用的。使用这两个宏的好处是使代码可读性提高，when 能省去 if 里的 progn 结构，unless 省去条件为真子句需要的的 nil 表达式。

2.4.3 循环

循环使用的是 while 表达式。它的形式是：

```
(while condition
  body)
```

比如：

```
(defun factorial (n)
  (let ((res 1))
    (while (> n 1)
      (setq res (* res n)
```

```
            n (- n 1)))
    res))
(factorial 10)                          ; => 3628800
```

2.5 逻辑运算

条件的逻辑运算和其它语言都是很类似的，使用 and、or、not。and 和 or 也同样具有短路性质。很多人喜欢在表达式短时，用 and 代替 when，or 代替unless。当然这时一般不关心它们的返回值，而是在于表达式其它子句的副作用。比如 or 经常用于设置函数的缺省值，而 and 常用于参数检查：

```
(defun hello-world (&optional name)
  (or name (setq name "Emacser"))
  (message "Hello, %s" name))           ; => hello-world
(hello-world)                           ; => "Hello, Emacser"
(hello-world "Ye")                      ; => "Hello, Ye"

(defun square-number-p (n)
  (and (>= n 0)
       (= (/ n (sqrt n)) (sqrt n))))
(square-number-p -1)                    ; => nil
(square-number-p 25)                    ; => t
```

2.6 函数列表

```
(defun NAME ARGLIST [DOCSTRING] BODY...)
(defvar SYMBOL &optional INITVALUE DOCSTRING)
(setq SYM VAL SYM VAL ...)
(let VARLIST BODY...)
(let* VARLIST BODY...)
(lambda ARGS [DOCSTRING] [INTERACTIVE] BODY)
(progn BODY ...)
(if COND THEN ELSE...)
(cond CLAUSES...)
(when COND BODY ...)
(unless COND BODY ...)
(when COND BODY ...)
(or CONDITIONS ...)
(and CONDITIONS ...)
(not OBJECT)
```