第九章　变量

在此之前，我们已经见过 elisp 中的两种变量，全局变量和 let 绑定的局部变量。它们相当于其它语言中的全局变量和局部变量。

关于 let 绑定的变量，有两点需要补充的。当同一个变量名既是全局变量也是局部变量，或者用 let 多层绑定，只有最里层的那个变量是有效的，用 setq 改变的也只是最里层的变量，而不影响外层的变量。比如：

```
(progn
  (setq foo "I'm global variable!")
  (let ((foo 5))
    (message "foo value is: %S" foo)
    (let (foo)
      (setq foo "I'm local variable!")
      (message foo))
    (message "foo value is still: %S" foo))
  (message foo))
```

另外需要注意一点的是局部变量的绑定不能超过一定的层数，也就是说，你不能把 foo 用 let 绑定 10000 层。当然普通的函数是不可能写成这样的，但是递归函数就不一定了。限制层数的变量在 max-specpdl-size 中定义。如果你写的递归函数有这个需要的话，可以先设置这个变量的值。

emacs 有一种特殊的局部变量——buffer-local 变量。

9.1　buffer-local 变量

emacs 能有如此丰富的模式，各个缓冲区之间能不相互冲突，很大程度上要归功于 buffer-local 变量。

声明一个 buffer-local 的变量可以用 make-variable-buffer-local 或用 make-local-variable。这两个函数的区别在于前者是相当于在所有变量中都产生一个 buffer-local 的变量。而后者只在声明时所在的缓冲区内产生一个局部变量，而其它缓冲区仍然使用的是全局变量。一般来说推荐使用 make-local-variable。

为了方便演示，下面的代码我假定你是在*scratch* 缓冲区里运行。我使用另一个一般都会有的缓冲区*Messages* 作为测试。先介绍两个用到的函数（ with-current-buffer 其实是一个宏）。

with-current-buffer 的使用形式是：

```
(with-current-buffer buffer
  body)
```

其中 buffer 可以是一个缓冲区对象，也可以是缓冲区的名字。它的作用是使其中的 body 表达式在指定的缓冲区里执行。

get-buffer 可以用缓冲区的名字得到对应的缓冲区对象。如果没有这样名字的缓冲区会返回 nil。

下面是使用 buffer-local 变量的例子：

```
(setq foo "I'm global variable!")       ; => "I'm global variable!"
(make-local-variable 'foo)              ; => foo
foo                                     ; => "I'm global variable!"
(setq foo "I'm buffer-local variable!") ; => "I'm buffer-local variable!"
foo                                  ; => "I'm buffer-local variable!"
(with-current-buffer "*Messages*" foo)  ; => "I'm global variable!"
```

从这个例子中可以看出，当一个符号作为全局变量时有一个值的话，用 make-local-variable 声明为 buffer-local 变量时，这个变量的值还是全局变量的值。这时候全局的值也称为缺省值。你可以用 default -value 来访问这个符号的全局变量的值：

```
(default-value 'foo)                    ; => "I'm global variable!"
```

如果一个变量是 buffer-local，那么在这个缓冲区内使用用 setq 就只能用改变当前缓冲区里这个变量的值。setq-default 可以修改符号作为全局变量的值。通常在.emacs 里经常使用 setq-default，这样可以防止修改的是导入.emacs 文件对应的缓冲区里的 buffer-local 变量，而不是设置全局的值。

测试一个变量是不是 buffer-local 可以用 local-variable-p：

```
(local-variable-p 'foo)                           ; => t
(local-variable-p 'foo (get-buffer "*Messages*")) ; => nil
```

如果要在当前缓冲区里得到其它缓冲区的 buffer-local 变量可以用 buffer-local-value：

```
(with-current-buffer "*Messages*"
  (buffer-local-value 'foo (get-buffer "*scratch*")))
                   ; => "I'm buffer local variable!"
```

9.2 变量的作用域

我们现在已经学习这样几种变量：

- 全局变量
- buffer-local 变量
- let 绑定局部变量

如果还要考虑函数的参数列表声明的变量，也就是 4 种类型的变量。那这种变量的作用范围(scope)和生存期（ extent）分别是怎样的呢？

作用域（ scope）是指变量在代码中能够访问的位置。emacs lisp 这种绑定称为 indefinite scope。indefinite scope 也就是说可以在任何位置都可能访问一个变量名。而 lexical scope（词法作用域）指局部变量只能作用在函数中和一个块里（ block）。

比如 let 绑定和函数参数列表的变量在整个表达式内都是可见的，这有别于其它语言词法作用域的变量。先看下面这个例子：

```
(defun binder (x)                    ; ‘x’ is bound in ‘binder’.
  (foo 5))                           ; ‘foo’ is some other function.
(defun user ()                       ; ‘x’ is used "free" in ‘user’.
  (list x))
(defun foo (ignore)
  (user))
(binder 10)                          ; => (10)
```

对于词法作用域的语言，在 user 函数里无论如何是不能访问 binder 函数中绑定的 x。但是在 elisp 中可以。

生存期是指程序运行过程中，变量什么时候是有效的。全局变量和 buffer-local 变量都是始终存在的，前者只能当关闭 emacs 或者用 unintern 从 obarray 里除去时才能消除。而 buffer-local 的变量也只能关闭缓冲区或者用 kill-local-variable 才会消失。而对于局部变量，emacs lisp 使用的方式称为动态生存期：只有当绑定了这个变量的表达式运行时才是有效的。这和 C 和 Pascal 里的 Local 和 automatic 变量是一样的。与此相对的是 indefinite extent，变量即使离开绑定它的表达式还能有效。比如：

```
(defun make-add (n)
  (function (lambda (m) (+ n m))))      ; Return a function.
(fset 'add2 (make-add 2))               ; Define function ‘add2’
                                        ;   with ‘(make-add 2)’.
(add2 4)                                ; Try to add 2 to 4.
```

其它 Lisp 方言中有闭包，但是 emacs lisp 中没有。

说完这些概念，可能你还是一点雾水。我给一个判断变量是否有效的方法吧：

1. 看看包含这个变量的 form 中是否有 let 绑定这个局部变量。如果这个 form 不是在定义一个函数，则跳到第 3 步。
2. 如果是在定义函数，则不仅要看这个函数的参数中是否有这个变量，而且还要看所有直接或间接调用这个函数的函数中是否有用 let 绑定或者参数列表里有这个变量名。这就没有办法确定了，所以你永远无法判断一个函数中出现的没有用 let 绑定，也不在参数列表中的变量是否是没有定义过的。但是一般来说这不是一个好习惯。
3. 看这个变量是否是一个全局变量或者是 buffer-local 变量。

对于在一个函数中绑定一个变量，而在另一个函数中还在使用，manual 里认为这两个种情况下是比较好的：

- 这个变量只有相关的几个函数中使用，在一个文件中放在一起。这个变量起程序里通信的作用。而且需要写好注释告诉其它程序员怎样使用它。
- 如果这个变量是定义明确、有很好文档作用的，可能让所有函数使用它，但是不要设置它。比如 case-fold-search。（我怎么觉得这里是用全局变量呢。）

思考题
先在*scratch* 缓冲区里运行了 (kill-local-variable 'foo) 后，运行几次下面的表达式，你能预测它们结果吗？

```
(progn
 (setq foo "I'm local variable!")
 (let ((foo "I'm local variable!"))
   (set (make-local-variable 'foo) "I'm buffer-local variable!")
   (setq foo "This is a variable!")
   (message foo))
 (message foo))
```

9.3 其它函数

一个符号如果值为空，直接使用可能会产生一个错误。可以用 boundp 来测试一个变量是否有定义。这通常用于 elisp 扩展的移植（用于不同版本或 XEmacs）。对于一个 buffer-local 变量，它的缺省值可能是没有定义的，这时用 default-value 函数可能会出错。这时就先用 default-boundp 先进行测试。

使一个变量的值重新为空，可以用 makunbound。要消除一个 buffer-local 变量用函数 kill-local-variable。可以用 kill-all-local-variables 消除所有的 buffer-local 变量。但是有属性 permanent-local 的不会消除，带有这些标记的变量一般都是和缓冲区模式无关的，比如输入法。

```
foo                                ; => "I'm local variable!"
(boundp 'foo)                      ; => t
(default-boundp 'foo)              ; => t
(makunbound 'foo)                  ; => foo
foo                                ; This will signal an error
(default-boundp 'foo)              ; => t
(kill-local-variable 'foo)         ; => foo
```

9.4 变量名习惯

对于变量的命名，有一些习惯，这样可以从变量名就能看出变量的用途：

- -hook 一个在特定情况下调用的函数列表，比如关闭缓冲区时，进入某个模式时。
- -function 值为一个函数
- -functions 值为一个函数列表
- -flag 值为 nil 或 non-nil

- -predicate 值是一个作判断的函数，返回 nil 或 non-nil
- -program 或-command 一个程序或 shell 命令名
- -form 一个表达式
- -forms 一个表达式列表。
- -map 一个按键映射（ keymap）

9.5 函数列表

```
(make-local-variable VARIABLE)
(make-variable-buffer-local VARIABLE)
(with-current-buffer BUFFER &rest BODY)
(get-buffer NAME)
(default-value SYMBOL)
(local-variable-p VARIABLE &optional BUFFER)
(buffer-local-value VARIABLE BUFFER)
(boundp SYMBOL)
(default-boundp SYMBOL)
(makunbound SYMBOL)
(kill-local-variable VARIABLE)
(kill-all-local-variables)
```

9.6 变量列表

```
max-specpdl-size
```

9.7 问题解答

9.7.1 同一个表达式运行再次结果不同?

运行第一次时，foo 缺省值为”I’m local variable!”，而 buffer-local 值为”This is a variable!”。第一个和第二个 message 都会显示”This is a variable!”。运行第二次时，foo 缺省值和 buffer-local 值都成了”I’m local variable!”，而第一次 message 显示”This is a variable!”，第二次显示”I’m local variable!”。这是由于 make-local-variable 在这个符号是否已经是 buffer-local 变量时有不同表现造成的。如果已经是一个 buffer-local 变量，则它什么也不做，而如果不是，则会生成一个 buffer-local 变量，这时在这个表达式内的所有 foo 也被重新绑定了。希望你写的函数能想到一点。