第6章 变 量

我们需要了解的下一个基本程序构造单元是变量。Common Lisp支持两种类型的变量：词法（lexical）变量和动态（dynamic）变量[①]。这两种变量类型分别对应于其他语言中的局部变量和全局变量，不过也只能说是大致相似。一方面，某些语言中的局部变量更像是Common Lisp的动态变量[②]。另一方面，某些语言中的局部变量虽然是词法作用域的，但并没有提供由Common Lisp的词法变量所提供的所有功能，尤其是并非所有语言都提供了支持闭包的词法作用域变量。

许多含有变量的表达式都可以同时使用词法变量和动态变量，这样一来更令人困惑了。因此本章我先讨论同时涉及两种类型的Lisp变量的几个方面，然后再谈及词法变量和动态变量各自的特征，随后再讨论Common Lisp的通用赋值操作符SETF，它用于为变量和其他任何可以保存值的位置赋予新值。

6.1 变量的基础知识

和其他语言一样，Common Lisp中的变量是一些可以保存值的具名位置。但在Common Lisp中，变量并非像在Java和C++等语言中那样带有确定的类型，也就是说不需要为每一个变量声明其可以保存对象的类型。相反，一个变量可以保存任何类型的值，并且这些值带有可用于运行期类型检查的类型信息。因此，Common Lisp是*动态类型*的——类型错误会被动态地检测到。举个例子，假如将某个并非数字的对象传给了+函数，那么Common Lisp将会报类型错误。而另一方面，Common Lisp是一种*强类型*语言，因为所有的类型错误都将被检测到——无法将一个对象作为其不属于的类型的实例来对待。[③]

---

① 出于某种你将在本章后面看到的原因，动态变量有时也称为*特殊变量*（special variable）。你需要注意这两个同义词，因为某些人（以及某些Lisp实现）使用其中一种术语，而其他人则习惯使用另外一种。

② 对于早期的Lisp而言，至少当其解释执行代码时，它倾向于使用动态变量作为局部变量。Elisp，用于Emacs的Lisp方言，在这方面有点守旧，它一直只支持动态变量。其他语言已经作出了从动态到词法的转换。例如Perl的`local`变量是动态的，而它由Perl 5所引入的my变量则是词法的。Python从来没有真正的动态变量，而只是在版本2.2以后引入了真正的词法作用域。（Python的词法变量相比Lisp而言仍在某些方面受限，因为语言语法中合并了赋值与绑定。）

③ 事实上，说所有类型错误将总能被检测并不是很正确——有可能使用可选的声明来告诉编译器，特定的变量将总是包含一个特定类型的对象，并且在一个特定的代码区域里关闭运行期类型检查。尽管如此，这类声明通常用于在开发和调试之后进行的代码优化，而不是在正常开发的期间。

至少从概念上来说，Common Lisp中所有的值都是对象的引用。[1]因此，将一个变量赋予新值就会改变该变量所指向的对象，而对之前被引用的对象却没有影响。尽管如此，如果一个变量保存了对一个可变对象的引用，那么就可以用该引用来修改此对象，而这种改动将应用于任何带有相同对象引用的代码。

而另一种已经用到的引入新变量的方式是定义函数形参。正如前一章所示，在用**DEFUN**来定义函数时，形参列表定义了当函数被调用时用来保存实参的变量。例如，下列函数定义了三个变量x、y和z，用来保存其实参：

```
(defun foo (x y z) (+ x y z))
```

每当函数被调用时，Lisp就会创建新的绑定来保存由函数调用者所传递的实参。绑定代表了变量在运行期的存在。单个变量就是可以在程序源代码中所指出的那种东西。在程序运行过程中可以有多个不同的绑定，单个变量甚至可以同时带有多重绑定。例如，一个递归函数的形参会在每一次函数调用中被重新绑定。

和所有Common Lisp变量一样，函数形参也可以保存对象引用。[2]因此，可以在函数体内为一个函数形参赋予新值，而这并不会影响到同样函数的另一个调用所创建的绑定。但如果改变了传递给函数的可变对象，则这些改动将会被调用者看到，因为无论调用者还是被调用者都在引用同一个对象。

引入新变量的另一种方式是使用**LET**特殊操作符。下面就是一个**LET**形式的结构：

```
(let (variable*)
  body-form*)
```

其中每一个*variable*都是一个变量初始化形式。每一个初始化形式要么是一个含有变量名和初值形式的列表，要么就是一个简单的变量名——作为将变量初始化到**NIL**的简略写法。例如，下面的**LET**形式会将三个变量x、y和z绑定到初始值10、20和**NIL**上：

```
(let ((x 10) (y 20) z)
  ...)
```

当这个**LET**形式被求值时，所有的初始值形式都将首先被求值，然后创建出新的绑定，并在形式体被执行之前这些绑定将初始化到适当的初始值上。在**LET**形式体中，变量名将引用新创建的绑定。在**LET**形式体执行结束后，这些变量名将重新引用在执行**LET**之前它们所引用的内容，如果有的话。

形式体中最后一个表达式的值将作为**LET**表达式的值返回。和函数形参一样，由**LET**所引入

① 作为一种优化，特定类型的对象，诸如特定大小以下的整数与字符，可能会在内存中直接表示，而其他对象将被表示成一个实际对象的指针。但由于整数和字符都是不可修改的，因此在不同变量中是否存在同一个对象的多个副本也就无关紧要了，这就是在第4章里所讨论的**EQ**和**EQL**的根本区别。

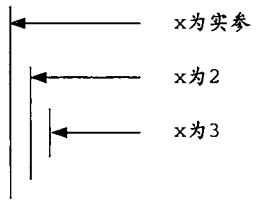
② 用编译器作者的话来说，Common Lisp函数是“传值的”，但被传递的值是对象的引用。这跟Java和Python的工作方式相似。

的变量将在每次进入LET时被重新绑定。[1]

函数形参和LET变量的作用域（变量名可用来引用该绑定的程序区域）被限定在引入该变量的形式之内，该形式即函数定义或LET，被称为绑定形式。你很快将看到，词法变量和动态变量使用两种略有不同的作用域机制，但两者的作用域都被界定在绑定形式之内。

如果嵌套了引入同名变量的绑定形式，那么最内层的变量绑定将覆盖外层的绑定。例如，在调用下面的函数时，将创建一个形参x的绑定来保存函数的参数。第一个LET创建了一个带有初始值2的新绑定，而内层的LET创建了另外一个绑定，其初始值为3。右边的竖线标记出了每一个绑定的作用域。

```
(defun foo (x)
  (format t "Parameter: ~a~%" x)
  (let ((x 2))
    (format t "Outer LET: ~a~%" x)
    (let ((x 3))
      (format t "Inner LET: ~a~%" x))
    (format t "Outer LET: ~a~%" x))
  (format t "Parameter: ~a~%" x))
```



每一次对x的引用都将指向最小封闭作用域中的绑定。一旦程序控制离开了一个绑定形式的作用域，其最近的闭合作用域中的绑定就被解除覆盖，并且x将转而指向它。因此，调用foo将得到这样的输出：

```
CL-USER> (foo 1)
Parameter: 1
Outer LET: 2
Inner LET: 3
Outer LET: 2
Parameter: 1
NIL
```

后面的章节将讨论其他可作为绑定形式使用的程序构造，其特点在于所引入的新变量名只能用于该构造。

例如，第7章将遇到DOTIMES循环，这是一种基本的计数循环。它引入了一个变量用来保存每次通过循环时递增的计数器的值。例如下面这个可以打印从0-9数字的循环，它绑定了变量x：

```
(dotimes (x 10) (format t "~d " x))
```

另一个绑定形式是LET的变体：LET*。两者的区别在于，在一个LET中，被绑定的变量名只能用在LET的形式体之内——LET形式中变量列表之后的那部分；但在一个LET*中，每个变量的初始值形式，都可以引用到那些在变量列表中早先引入的变量。因此可以写成下面这样：

---

① LET形式和函数形参中的变量是以完全相同的机制创建的。事实上，在某些Lisp方言中，尽管不是Common Lisp，LET只是一个展开到一个匿名函数调用的宏。也就是说，在那些方言里，

```
(let ((x 10)) (format t "~a" x))
```

是一个展开到下列结果的宏：

```
((lambda (x) (format t "~a" x)) 10)
```

```
(let* ((x 10)
       (y (+ x 10)))
  (list x y))
```

但不能这样写：

```
(let ((x 10)
      (y (+ x 10)))
  (list x y))
```

不过也可以通过嵌套的**LET**来达到相同的效果：

```
(let ((x 10))
  (let ((y (+ x 10)))
    (list x y)))
```

6.2　词法变量和闭包

默认情况下，Common Lisp中所有的绑定形式都将引入*词法作用域*变量。词法作用域的变量只能由那些在文本上位于绑定形式之内的代码所引用。那些曾经使用Java、C、Perl或者Python来编程的人们应该熟悉词法作用域，因为它们都提供词法作用域的局部变量。如此说来，Algol程序员们也该对其感到自然才是，因为Algol在20世纪60年代首先引入了词法作用域。

尽管如此，但Common Lisp的词法变量还是有一些变化的，至少和最初的Algol模型相比是这样。变化之处在于将词法作用域和嵌套函数一起使用时，按照词法作用域的规则，只有文本上位于绑定形式之内的代码可以指向一个词法变量。但是当一个匿名函数含有一个对来自封闭作用域之内词法变量的引用时，将会发生什么呢？例如，在下面的表达式中：

```
(let ((count 0)) #'(lambda () (setf count (1+ count))))
```

根据词法作用域规则，**LAMBDA**形式中对count的引用应该是合法的，而这个含有引用的匿名函数将被作为**LET**形式的值返回，并可能会通过**FUNCALL**被不在**LET**作用域之内的代码所调用。这样会发生什么呢？正如你将看到的那样，当count是一个词法变量时，情况一切正常。本例中，当控制流进入**LET**形式时所创建的count绑定将被尽可能地保留下来，只要某处保持了一个对**LET**形式所返回的函数对象的引用即可。这个匿名函数被称为一个*闭包*，因为它“封闭包装”了由**LET**创建的绑定。

理解闭包的关键在于，被捕捉的是绑定而不是变量的值。因此，一个闭包不仅可以访问它所闭合的变量的值，还可以对其赋予在闭包被调用时不断变化的新值。例如，可以像下面这样将前面的表达式所创建的闭包捕捉到一个全局变量里：

```
(defparameter *fn* (let ((count 0)) #'(lambda () (setf count (1+ count)))))
```

然后每当调用它时，count的值将被加1：

```
CL-USER> (funcall *fn*)
1
CL-USER> (funcall *fn*)
2
```

```
CL-USER> (funcall *fn*)
3
```

单一闭包可以简单地通过引用变量来闭合许多变量绑定，或是多个闭合可以捕捉相同的绑定。例如，下面的表达式返回由三个闭合所组成的列表，一个可以递增其所闭合的count绑定的值，另一个可以递减它，还有一个返回它的当前值。

```
(let ((count 0))
  (list
   #'(lambda () (incf count))
   #'(lambda () (decf count))
   #'(lambda () count)))
```

6.3 动态变量

词法作用域的绑定通过限制作用域（其中给定的名字只具有字面含义）使代码易于理解，这就是大多数现代语言将词法作用域用于局部变量的原因。尽管如此，有时的确需要全局变量——一种可以从程序的任何位置访问到的变量。尽管随意使用全局变量将使代码变得杂乱无章，就像毫无节制地使用goto那样，但全局变量确实有其合理的用途，并以某种形式存在于几乎每种编程语言里。[①]正如你即将看到的，Lisp的全局变量和动态变量都更为有用并且更易于管理。

Common Lisp提供了两种创建全局变量的方式：**DEFVAR**和**DEFPARAMETER**。两种形式都接受一个变量名、一个初始值以及一个可选的文档字符串。在被**DEFVAR**和**DEFPARAMETER**定义以后，该名字可用于任何位置来指向全局变量的当前绑定。如前所述，全局变量习惯上被命名为以*开始和结尾的名字。你将在本节后面看到所述遵守该命名约定的重要性。**DEFVAR**和**DEFPARAMETER**的示例如下：

```
(defvar *count* 0
  "Count of widgets made so far.")

(defparameter *gap-tolerance* 0.001
  "Tolerance to be allowed in widget gaps.")
```

两种形式的区别在于，**DEFPARAMETER**总是将初始值赋给命名的变量，而**DEFVAR**只有当变量未定义时才这样做。**DEFVAR**形式也可以不带初始值来使用，从而在不给定其值的情况下定义一个全局变量。这样的变量称为未绑定的（unbound）。

从实践上来讲，应该使用**DEFVAR**来定义某些变量，这些变量所含数据是应持久存在的，即使用到该变量的源码发生改变时也应如此。例如，假设前面定义的两个变量是一个用来控制部件工厂的应用程序的一部分，那么使用**DEFVAR**来定义*count*变量就比较合适，因为目前已生产的部件数量不会因为对部件生产的代码做了某些改变而就此作废。[②]

① Java将全局变量伪装成公共静态字段，C使用extern变量，而Python的模块级别变量和Perl的包级别变量同样可以从任何位置被访问到。

② 如果你特定想要重设由**DEFVAR**定义的变量，那要么使用**SETF**直接设置它，要么使用**MAKUNBOUND**先将其变成未绑定的，再重新求值**DEFVAR**形式。

另一方面，假定变量*gap-tolerance*对于部件生产代码本身的行为具有影响。如果你决定使用一个或紧或松的容差值，并且改变了**DEFPARAMETER**形式中的值，那么就要在重新编译和加载文件时让这一改变产生效果。

在用**DEFVAR**和**DEFPARAMETER**定义了一个变量之后，就可以从任何一个地方引用它。例如，可以定义下面的函数来递增已生产部件的数量。

```
(defun increment-widget-count () (incf *count*))
```

全局变量的优势在于不必到处传递它们。多数语言将标准输入与输出流保存在全局变量里正是出于这个原因——你根本不知道什么时候会向标准输出流打印东西，并且也不想仅仅由于日后有人需要，就使每个函数都不得不接受并传递含有这些流的参数。

不过，一旦像标准输出流这样的值被保存在一个全局变量中，并且已经编写了引用那个全局变量的代码，那么试图通过更改变量值来临时改变代码行为的做法就颇为诱人了。

例如，假设正工作的一个程序中含有的某些底层日志函数会将输出打印到位于全局变量*standard-output*中的流上。现在假设在程序的某个部分里，想要将所有这些函数所生成的输出捕捉到一个文件里，那么可以打开一个文件并将得到的流赋予*standard-output*。现在底层函数们将把它们的输出发往该文件。

这样工作得很好，但假如完成工作时忘记将*standard-output*重新设置回最初的流上，那么程序中所有用到*standard-output*的其他代码也会将把它们的输出发往该文件。[①]

真正所需的代码包装方式似乎应如下所述："在从这里以下的所有代码中——所有它调用的函数以及它们进一步调用的函数，诸如此类，直到最底层的函数全局变量*standard-output*都应使用该值。"然后当上层的函数返回时，*standard-output*应该自动恢复到其原来的值。

这看起来正像是Common Lisp的另一种变量，即动态变量所做的事。当绑定了一个动态变量时，例如通过一个**LET**变量或函数形参，在被绑定项上所创建的绑定替换了在绑定形式期间的对应全局绑定。与词法绑定——只能被绑定形式的词法作用域之内的代码所引用——所不同的是，动态绑定可以被任何在绑定形式执行期间所调用到的代码所引用。[②]显然所有全局变量事实上都是动态变量。

因此，如果想要临时重定义*standard-output*，只需重新绑定它即可，比如说可以使用**LET**。

---

① 这种临时重新赋值* standarel-output *的策略也会在系统使用多线程时失效——如果有多个控制线程同时试图输出到不同的流上，它们将全都试图设置该全局变量到它们想要使用的流上，完全无视彼此的感受。你可以使用一个锁来控制对一个全局变量的访问，但那样就无法充分获得多重并发线程所带来的好处了。因为无论哪个线程正在输出时，即使它们想要输出到一个不同的流上，它将不得不锁住所有其他线程直到完成。

② 一个绑定可被引用到的时间间隔，其技术术语称为生存期，这样作用域（scope）和生存期（extent）就是两个紧密相关的概念——作用域关注空间而生存期关注时间。词法变量具有词法作用域和不确定的（indefinite）生存期，意思是它们可以在不定长的间隔里保持存在，这取决于它们被需要多久。动态变量正好相反，它们具有不确定的作用域，因为它们可从任何位置访问却有动态的生存期。更加引起误会的是，不确定作用域和动态生存期的组合经常被错误地称为动态作用域（dynamic scope）。

```
(let ((*standard-output* *some-other-stream*))
  (stuff))
```

在任何由于调用stuff而运行的代码中，对*standard-output*的引用将使用由**LET**所建立的绑定，并且当stuff返回并且程序控制离开**LET**时，这个对*standard-output*的新绑定将随之消失，接下来对*standard-output*的引用将看到**LET**之前的绑定。在任何给定时刻，最近建立的绑定会覆盖所有其他的绑定。从概念上讲，一个给定动态变量的每个新绑定都将被推到一个用于该变量的绑定栈中，而对该变量的引用总是使用最近的绑定。当绑定形式返回时，它们所创建的绑定会被从栈上弹出，从而暴露出前一个绑定。[①]

一个简单的例子就能揭示其工作原理。

```
(defvar *x* 10)
(defun foo () (format t "X: ~d~%" *x*))
```

上面的**DEFVAR**为变量*x*创建了一个到数值10的全局绑定。函数foo中，对*x*的引用将动态地查找其当前绑定。如果从最上层调用foo，由**DEFVAR**所创建的全局绑定就是唯一可用的绑定，因此它打印出10：

```
CL-USER> (foo)
X: 10
NIL
```

但也可以用**LET**创建一个新的绑定来临时覆盖全局绑定，这样foo将打印一个不同的值：

```
CL-USER> (let ((*x* 20)) (foo))
X: 20
NIL
```

现在不使用**LET**再次调用foo，它将再次看到全局绑定：

```
CL-USER> (foo)
X: 10
NIL
```

现在定义另一个函数：

```
(defun bar ()
  (foo)
  (let ((*x* 20)) (foo))
  (foo))
```

注意，中间那个对foo的调用被包装在一个将*x*绑定到新值20的**LET**形式中。运行bar得到的结果如下所示：

```
CL-USER> (bar)
X: 10
X: 20
X: 10
NIL
```

① 尽管标准并未指定如何在Common Lisp中使用多线程，但所有提供多线程的实践都遵循了由Lisp机所建立的原则，在每线程的基础上创建动态绑定，一个对全局变量的引用将查找当前线程中最近建立的绑定，或是全局绑定。

可以看到，第一次对foo的调用看到了全局绑定，其值为10。然而，中间的那个调用却看到了新的绑定，其值为20。但在**LET**之后，foo再次看到了全局绑定。

和词法绑定一样，赋予新值仅会影响当前绑定。为了理解这点，可以重定义foo来包含一个对*x*的赋值。

```
(defun foo ()
  (format t "Before assignment~18tX: ~d~%" *x*)
  (setf *x* (+ 1 *x*))
  (format t "After assignment~18tX: ~d~%" *x*))
```

现在foo打印*x*的值，对其递增，然后再次打印它。如果你只运行foo，将看到这样的结果：

```
CL-USER> (foo)
Before assignment X: 10
After assignment  X: 11
NIL
```

这看起来很正常，现在运行bar：

```
CL-USER> (bar)
Before assignment X: 11
After assignment  X: 12
Before assignment X: 20
After assignment  X: 21
Before assignment X: 12
After assignment  X: 13
NIL
```

注意*x*从11开始——之前的foo调用真的改变了全局的值。来自bar的第一次对foo的调用将全局绑定递增到12。中间的调用由于**LET**的关系没有看到全局绑定，然后最后一个调用再次看到了全局绑定，并将其从12递增到13。

那么它是怎样工作的呢？**LET**是怎样知道在它绑定*x*时打算创建的是动态绑定而不是词法绑定呢？这是因为该名字已经被声明为特殊的（special）。[①]每一个由**DEFVAR**和**DEFPARAMETER**所定义的变量其名字都将被自动声明为全局特殊的。这意味着无论何时你在绑定形式中使用这样一个名字，无论是在**LET**中，或是作为一个函数形参，亦或是在任何创建新变量绑定的构造中，被创建的绑定将成为一个动态绑定。这就是为什么命名约定如此重要——如果你使用了一个变量，以为它是词法变量，而它却刚好是全局特殊的变量，这就很不好。一方面，你所调用的代码可能在你意想之外改变了绑定的值；另一方面，你可能会覆盖一个由栈的上一级代码所建立的绑定。如果总是按照*命名约定来命名全局变量，就不会在打算建立词法绑定时却意外使用了动态绑定。

也有可能将一个名字声明为局部特殊的，如果在一个绑定形式里将一个名字声明为特殊的，那么为该变量所创建的绑定将是动态的而不是词法的。其他代码可以局部地声明一个名字为特殊的，从而指向该动态绑定。尽管如此，局部特殊变量使用相对较少，所以你不需要担心它们。[②]

① 这就是动态变量有时也被称为特殊变量的原因。

② 如果你一定想知道的话，你可以在HyperSpec上查找**DECLARE**、**SPECIAL**和**LOCALLY**。

动态绑定使全局变量更易于管理，但重要的是注意到它们将允许超距作用的存在。绑定一个全局变量具有两种超距效果——它可以改变下游代码的行为，并且它也开启了一种可能性，使得下游代码可以为栈的上一级所建立的绑定赋予一个新的值。你应该只有在需要利用这两个特征时才使用动态变量。

6.4 常量

我尚未提到的另一种类型的变量是常值变量（constant variable）。所有的常量都是全局的，并且使用**DEFCONSTANT**定义，**DEFCONSTANT**的基本形式与**DEFPARAMETER**相似。

```
(defconstant name initial-value-form [ documentation-string ])
```

与**DEFVAR**和**DEFPARAMETER**相似，**DEFCONSTANT**在所使用的名字上产生了全局效果——从此该名字仅被用于指向常量，它不能被用作函数形参或是用任何其他的绑定形式进行重绑定。因此，许多Lisp程序员遵循了一个命名约定，用以+开始和结尾的名字来表示常量，这一约定在某种程度上不像全局特殊名字的*命名约定那样流行，但也不错。[1]

关于**DEFCONSTANT**，需要注意的另一点是，尽管语言允许通过重新求值一个带有初始值形式的**DEFCONSTANT**来重定义一个常量，但在重定义之后究竟发生什么是没有定义的。在实践上，多数实现将要求对任何引用了该常量的代码进行求值以便它们能看到新值，因为老的值可能已经内联到代码中了。因此最好只用**DEFCONSTANT**来定义那些真正是常量的东西，例如π。而对于那些可能想改变的东西，则应转而使用**DEFPARAMETER**。

6.5 赋值

一旦创建了绑定，就可以对它做两件事：获取当前值以及为它设置新值。正如在第4章里所看到的，一个符号被求值为它所命名的变量的值，因此，可以简单地通过引用这个变量来得到它的当前值。而为绑定赋予新值则要使用**SETF**宏——Common Lisp的通用赋值操作符。下面是**SETF**的基本形式：

```
(setf place value)
```

因为**SETF**是宏，所以它可以检查它所赋值的*place*上的形式，并展开成适当的底层操作来修改那个位置。当该位置是变量时，它展开成一个对特殊操作符**SETQ**的调用，后者可以访问到词法和动态绑定。[2]例如，为了将值10赋给变量x，可以写成这样：

```
(setf x 10)
```

正如早先所讨论的，为一个绑定赋予新值对该变量的任何其他绑定没有效果，并且它对赋值之前绑定上所保存的值也没有任何效果。因此，函数

① 一些由语言本身所定义的关键常量并不遵循这一约定，包括但不限于**T**和**NIL**，这在偶尔有人想用t作为局部变量名时会很讨厌。另一个是**PI**，其含有最接近数学常量π的长浮点值。

② 某些守旧的Lisp程序员喜欢使用**SETQ**对变量赋值，但现代风格倾向于将**SETF**用于所有的赋值操作。

```
(defun foo (x) (setf x 10))
```

中的**SETF**对于foo之外的任何值都没有效果。这个在foo被调用时所创建的绑定被设置到10，立即替换了作为参数传递的任何值，特别是在如下形式中：

```
(let ((y 20))
  (foo y)
  (print y))
```

将打印出20而不是10，因为传递给foo的y的值在该函数中变成了x的值，随后又被**SETF**设置成新值。

SETF也可用于依次对多个位置赋值。例如，与其像下面这样：

```
(setf x 1)
(setf y 2)
```

也可写成：

```
(setf x 1 y 2)
```

SETF返回最近被赋予的值，因此也可以像下面的表达式那样嵌套调用**SETF**，将x和y赋予同一个随机值：

```
(setf x (setf y (random 10)))
```

6.6　广义赋值

当然，变量绑定并不是唯一可以保留值的位置，Common Lisp还支持复合数据结构，包括数组、哈希表、列表以及由用户定义的数据结构，所有这些都含有多个可用来保存值的位置。

后续章节里将讨论那些数据结构，但就目前所讨论的赋值主题而言，你应该知道**SETF**可以为任何位置赋值。当描述不同的复合数据结构时，我将指出哪些函数可以作为**SETF**的“位置”来使用。总之，如果需要对位置赋值，那么几乎肯定要用到**SETF**。虽然在此不予介绍，但**SETF**经拓展后甚至可为由用户定义的位置赋值。[①]

从这个角度来说，**SETF**和多数源自C的语言中的赋值操作符没有区别。在那些语言里，=操作符可以将新值赋给变量、数组元素和类的字段。在诸如Perl和Python这类支持哈希表作为内置数据类型的语言里，=也可以设置哈希表项的值。表6-1总结了=在那些语言里的不同用法。

表6-1　= 在其他语言中的用法

赋值对象	Java, C, C++	Perl	Python
变量	x = 10;	$x = 10;	x = 10
数组元素	a[0] = 10;	$a[0] = 10;	a[0] = 10
哈希表项		$hash{'key'} = 10;	hash['key'] = 10
对象字段	o.field = 10;	$o->{'field'} = 10;	o.field = 10

① 查看**DEFSETF**和**DEFINE-SETF-EXPANDER**以获取进一步的信息。

SETF以同样的方式工作——SETF的第一个参数用来保存值的位置，而第二个参数提供了值。和这些语言中的=操作符一样，你可以使用和正常获取其值相同的形式来表达位置。[1] 因此，表6-1中赋值语句的Lisp等价形式分别为：AREF是数组访问函数，GETHASH做哈希表查找，而field可能是一个访问某用户定义对象中名为field的成员的函数。如下所示：

```
Simple variable:     (setf x 10)
Array:               (setf (aref a 0) 10)
Hash table:          (setf (gethash 'key hash) 10)
Slot named 'field': (setf (field o) 10)
```

注意，当用SETF对一个作为更大对象一部分的位置进行赋值时，与赋值一个变量具有相同的语义：被修改的位置对之前保存在该位置上的对象没有任何影响。再次说明，这跟=在Java、Perl和Python中的行为非常相似。[2]

6.7 其他修改位置的方式

尽管所有的赋值都可以用SETF来表达，但有些固定模式（比如像基于当前值来赋予新值）由于经常使用，因此有它们自己的操作符。例如，尽管可以像这样使用SETF来递增一个数：

```
(setf x (+ x 1))
```

或是像这样来递减它：

```
(setf x (- x 1))
```

但跟C风格的++x和--x相比就显得很冗长了。相反，可以使用宏INCF和DECF，它们以默认为1的特定数量对一个位置的值进行递增和递减。

```
(incf x)     ≡ (setf x (+ x 1))
(decf x)     ≡ (setf x (- x 1))
(incf x 10) ≡ (setf x (+ x 10))
```

类似INCF和DECF这种宏称为修改宏（modify macro），修改宏是建立在SETF之上的宏，其基于作用位置上的当前值来赋予该位置一个新值。修改宏的主要好处是，它们比用SETF写出的同样的修改语句更加简洁。另外，修改宏所定义的方式使其可以安全地用于那些表达式必须只被求值一次的位置。在下面这个表达式中，INCF会递增一个数组中任意元素的值，这个例子确实很可笑：

```
(incf (aref *array* (random (length *array*))))
```

一个到SETF表达式的幼稚转换方法可能看起来像这样：

① 源自Algol，使用=左边的“位置”和右边的新值进行赋值的语法得到广泛使用，由此产生了术语“左值”（lvalue，left value的缩写），意思是可被赋值的某种东西，以及“右值”（rvalue），意思是某种可以提供值的东西。一个编译器黑客将会说，“SETE将其第一个参数视为左值”。

② C程序员可能将变量和其他位置看成是保存了实际对象的指针。对一个变量赋值简单地改变了其所指向的对象，而对一个复合对象中的一部分赋值，则类似于重定向通向实际对象的指针。C++程序员应该注意到在C++中当处理对象时，确切地说，在进行成员复制时，=的行为是相当特异的。

```
(setf (aref *array* (random (length *array*)))
      (1+ (aref *array* (random (length *array*)))))
```

但它不会正常工作，因为两次对**RANDOM**的调用不一定能返回相同的值——该表达式将很可能抓取数组中一个元素的值，将其递增，然后将其作为新值保存到另一个不同的数组元素上。与之相比，上面的INCF表达式却能产生正确的行为，因为它知道如何处理这个表达式：

```
(aref *array* (random (length *array*)))
```

取出其中可能带有副作用的部分，从而确保它们仅被求值一次。在本例中，经展开后，它差不多会等价于以下形式。

```
(let ((tmp (random (length *array*))))
  (setf (aref *array* tmp) (1+ (aref *array* tmp))))
```

一般而言，修改宏可以保证以从左到右的顺序，对它们的参数和位置形式的子形式每个只求值一次。

在第3章那个微型数据库中曾用来向*db*变量添加元素的宏**PUSH**则是另一个修改宏。第12章在讲到如何在Lisp中表示列表时会详细地介绍它及其对应的**POP**和**PUSHNEW**是如何工作的。

最后有两个稍微有些难懂但很有用的修改宏，它们是**ROTATEF**和**SHIFTF**。**ROTATEF**在位置之间轮换它们的值。如果有两个变量a和b，那么如下调用

```
(rotatef a b)
```

将交换两个变量的值并返回**NIL**。由于a和b是变量并且不需要担心副作用，因此前面的ROTATEF的表达式将等价于如下形式：

```
(let ((tmp a)) (setf a b b tmp) nil)
```

对于其他不同类型的位置，使用**SETF**的等价表达式将会更加复杂一些。

SHIFTF与之相似，除了它将值向左侧移动而不是轮换它们——最后一个参数提供的值移动到倒数第二个参数上，而其他的值将向左移动一个，第一个参数的最初值将被简单地返回。这样，

```
(shiftf a b 10)
```

将等价于如下形式。同样，不必担心副作用。

```
(let ((tmp a)) (setf a b b 10) tmp)
```

ROTATEF和**SHIFTF**都可被用于任意多个参数，并且和所有的修改宏一样，它们可以保证以从左到右的顺序对每个参数仅求值一次。

学完了Common Lisp函数和变量的基础知识以后，下面将开始介绍一个令Lisp始终区别于其他语言的特性：宏。