第7章 宏：标准控制构造

尽管起源于Lisp的许多编程思想，从条件表达式到垃圾收集，都已经被吸取进其他语言，但Lisp的宏系统却始终使它保持了在语言风格上的独特性。遗憾的是，宏这个字虽然在计算领域可以描述很多东西，但和Common Lisp的宏相比，它们仅大致相似。当Lisp程序员们试图向非Lisp程序员解释宏这种特性的伟大之处时，这种相似性导致了无休止的误解。[1]要想理解Lisp的宏，就真的需要重新看待它，不能带有任何基于其他碰巧也叫做宏的概念所带来的成见。现在先退一步，从观察各种语言支持扩展的不同方式讨论Lisp宏。

所有的程序员应该都熟知这么一种观点，语言的定义可能会包含一个借由“核心”语言实现的标准功能库——如果某些功能没有定义在标准库中，那么它们可能已经被程序员实现在语言中了。只要它还没有被定义成标准库的一部分，那么这些功能就可以被任何程序员在语言之上实现。例如，C的标准库就差不多可以完全用可移植的C来实现。类似地，Java的标准Java开发包（JDK）中所提供的不断改进的类和接口集合也是用“纯”Java编写的。

使用核心加上标准库的方式来定义语言的优势在于易于理解和实现。但真正的好处在于其可表达性——由于所认为的“语言”很大程度上其实是一个库，因此很容易对其进行扩展。如果C语言中不含有所需的用来做某件事的一个函数，那就可以写出这个函数，然后就得到了一个特性稍微丰富一点的C版本。类似地，在诸如Java或Smalltalk这类几乎所有有趣部分都是由类来定义的语言里，通过定义新的类就可以扩展该语言，使其更适用于编写你正试图编写的任何程序。

尽管Common Lisp支持所有这些扩展语言的方法，但宏还提供了另一种方式。如同第4章所述，每个宏都定义了自己的语法，它们能够决定那些被传递的S-表达式如何转换成Lisp形式。核心语言有了宏，就有可能构造出新的语法，诸如`WHEN`、`DOLIST`和`LOOP`这样的控制构造以及`DEFUN`和`DEFPARAMETER`这样的定义形式，从而使这些新语法可以作为“标准库”的一部分而不是将其硬编码到语言核心。这已经牵涉到语言本身是如何实现的，但作为一个Lisp程序员，你更关心的将是它所提供的另一种语言扩展方式，而这将使Common Lisp成为更好的用于表达特定编程问题

① 想要了解这类误解，可以在相对长期的Usenet新闻组上，以macro为主题，在comp.lang.lisp和comp.lang.*之间交叉投递的讨论中搜索。一个大致的说法如下：
Lisp支持者：“Lisp是最强的，因为它有宏！”
其他语言支持者：“你认为Lisp好是因为它的宏吗？！但宏是可怕和有害的，因此Lisp也一定是可怕和有害的。”

解决方案的语言。

那么，利用另一种方式来拓展语言的好处似乎是显而易见的。但出于某些原因，对于大量没有实际使用过Lisp宏的人，他们可以为了解决编程问题而日复一日地去创建新的函数型抽象或定义类的层次体系，却被这种可以定义新的句法抽象的思想给吓到了。通常，这种宏恐惧症的原因多半是来自学习其他“宏”系统时的不良经历。简单地对未知事物的恐惧无疑也是其中一部分原因。为了避免触发任何宏恐惧症反应，我们将从Common Lisp所定义的几种标准控制构造宏开始讨论，既而缓慢进入该主题。它们都是那些如果Lisp没有宏，就必须构造在语言核心里的东西。使用它们时，你不必在意它们是作为宏实现的，尽管如此，它们的确可以很好地展示出宏的一些功用。[1]下一章将说明如何定义你自己的宏。

7.1 WHEN 和 UNLESS

如前所述，最基本的条件执行形式是由**IF**特殊操作符提供的，其基本形式是：如果*x*成立，那么执行*y*；否则执行*z*。

```
(if condition then-form [else-form])
```

*condition*被求值，如果其值非**NIL**，那么*then-form*会被求值并返回其结果。否则，如果有*else-form*，它将被求值并返回其结果。如果*condition*是**NIL**并且没有*else-form*，那么**IF**返回**NIL**。

```
(if (> 2 3) "Yup" "Nope") → "Nope"
(if (> 2 3) "Yup")        → NIL
(if (> 3 2) "Yup" "Nope") → "Yup"
```

尽管如此，**IF**事实上并不是什么伟大的句法构造，因为每个*then-form*和*else-form*都被限制必须是单一的Lisp形式。这意味着如果想在每个子句中执行一系列操作，则必须将其用其他一些语法形式进行封装。举个例子，假如在一个垃圾过滤程序中，当一个消息是垃圾时，你想要在将其标记为垃圾的同时更新垃圾数据库，那么你不能这样写：

```
(if (spam-p current-message)
    (file-in-spam-folder current-message)
    (update-spam-database current-message))
```

因为对update-spam-database的调用将被作为**else**子句来看待，而不是**then**子句的一部分。另一个特殊操作符**PROGN**可以按顺序执行任意数量的形式并返回最后一个形式的值。因此可以通过写成下面这样来得到预想的行为：

```
(if (spam-p current-message)
    (progn
      (file-in-spam-folder current-message)
      (update-spam-database current-message)))
```

① 另一个重要的用宏来定义的语言构造类别是所有的定义性构造，诸如**DEFUN**、**DEFPARAMETER**以及**DEFVAR**。在第24章里你将定义自己的定义性宏，从而可以简洁地编写可以用来读写二进制数据的代码。

这样做并不算太坏。但假如不得不多次使用这样的写法，不难想象你将在一段时间以后开始厌倦它。你可能会自问："为什么Lisp没有提供一种方式来做我真正想做的事，也就是说，'当x为真时，做这个、那个以及其他一些事情'？"换句话说，你很快将注意到这种由IF加上PROGN所组成的模式，并且希望可以有一种方式来抽象所有细节而不是每次都将它们写出来。

这正是宏所能够提供的功能。在这个案例中，Common Lisp提供了一个标准宏WHEN，可以让你写成这样：

```
(when (spam-p current-message)
  (file-in-spam-folder current-message)
  (update-spam-database current-message))
```

如果它没有被内置到标准库中，你也可以像下面这样用一个宏来自己定义WHEN，这里用到了第3章中讨论过的反引号：[①]

```
(defmacro when (condition &rest body)
  `(if ,condition (progn ,@body)))
```

与WHEN宏同系列的另一个宏是UNLESS，它取相反的条件，只有当条件为假时才求值其形式体。换句话说：

```
(defmacro unless (condition &rest body)
  `(if (not ,condition) (progn ,@body)))
```

必须承认，这些都是相当简单的宏。这里没有什么高深的道理，它们只是抽象掉了一些语言层面约定俗成的细节，从而允许你更加清晰地表达你的真实意图。它们的极度简单性产生了一个重要的观点：由于宏系统是直接构建在语言之中的，所以可以写出像WHEN和UNLESS这样简单的宏来获得小而重要的清晰性，并随后通过不断地使用而无限放大。第24、26和31章将展现宏是如何被更大规模地用于创建完整的特定领域的嵌入式语言。首先来介绍一下标准控制构造宏。

7.2 COND

当遇到多重分支的条件语句时，原始的IF表达式再一次变得丑陋不堪：如果a成立那么执行x，否则如果b成立那么执行y；否则执行z。只用IF来写这样的条件表达式链并没有逻辑问题，只是不太好看。

```
(if a
    (do-x)
    (if b
       (do-y)
       (do-z)))
```

如果需要在then子句中包括多个形式，那就需要用到PROGN，而那样事情就会变得更糟。因此毫不奇怪，Common Lisp提供了一个用于表达多重分支条件的宏COND。下面是它的基本结构：

① 其实不能将这个定义输入到Lisp中，因为重定义WHEN所在的COMMON-LISP包中的名字是非法的。如果你真想写出这样一个宏，则需要改变名字，例如my-when。

```
(cond
  (test-1 form*)
      .
      .
      .
  (test-N form*))
```

主体中的每个元素都代表一个条件分支，并由一个列表所构成，列表中含有一个条件形式，以及零或多个当该分支被选择时将被求值的形式。这些条件形式按照分支在主体中出现的顺序被依次求值，直到它们中的一个求值为真。这时，该分支中的其余形式将被求值，且分支中最后一个形式的值将作为整个COND的返回值。如果该分支在条件形式之后不再含有其他形式，那么就将返回该条件形式的值。习惯上，那个用来表示if/else-if链中最后一个else子句的分支将被写成带有条件T。虽然任何非NIL的值都可以使用，但在阅读代码时，T标记确实有用。这样就可以像下面这样用COND来写出前面的嵌套IF表达式：

```
(cond (a (do-x))
      (b (do-y))
      (t (do-z)))
```

7.3 AND、OR 和 NOT

在使用IF、WHEN、UNLESS和COND形式编写条件语句时，经常用到的三个操作符是布尔逻辑操作符AND、OR和NOT。

严格来讲，NOT这个函数并不属于本章讨论范畴，但它跟AND和OR紧密相关。它接受单一参数并对其真值取反，当参数为NIL时返回T，否则返回NIL。

AND和OR则是宏。它们实现了对任意数量子表达式的逻辑合取和析取操作，并被定义成宏以便支持“短路”特性。也就是说，它们仅以从左到右的顺序对用于检测整体真值的必要数量的子表达式进行求值。这样，只要AND的一个子表达式求值为NIL，它就立即停止并返回NIL。如果所有子表达式都求值到非NIL，那么它将返回最后一个子表达式的值。而对于OR来说，只要一个子表达式求值到非NIL，它就立即停止并返回当前子表达式的值。如果没有子表达式求值到真，OR返回NIL。下面是一些例子：

```
(not nil)              → T
(not (= 1 1))          → NIL
(and (= 1 2) (= 3 3))  → NIL
(or (= 1 2) (= 3 3))   → T
```

7.4 循环

循环构造是另外一类主要的控制构造。Common Lisp的循环机制，除了更加强大和灵活以外，还是一门关于宏所提供的“鱼和熊掌兼得”的编程风格的有趣课程。

初看起来，Lisp的25个特殊操作符中没有一个能够直接支持结构化循环，所有的Lisp循环控

制构造都是构建在一对提供原生goto机制的特殊操作符之上的宏。[1]和许多好的抽象或句法一样，Lisp的循环宏构建在以那两个特殊操作符为基础的一组分层抽象之上。

底层（不考虑特殊操作符）是一个非常通用的循环构造DO。尽管非常强大，但DO和许多其他的通用抽象一样，在应用于简单情形时显得过于复杂。因此Lisp还提供了另外两个宏，DOLIST和DOTIMES。它们不像DO那样灵活，却提供了对于常见的在列表元素上循环和计数循环的便利支持。尽管一个实现可以用任何方式来实现这些宏，但它们被典型实现为展开到等价DO循环的宏。因此，在由Common Lisp特殊操作符所提供的底层原语之上，DO提供了一种基本的结构化循环构造，而DOLIST和DOTIMES则提供了两种易用却不那么通用的构造。并且如同在下一章将看到的那样，对于那些DOLIST和DOTIMES无法满足需要的情形，还可以在DO之上构建自定义的循环构造。

最后，LOOP宏提供了一种成熟的微型语言，它用一种非Lisp的类似英语（或至少类似Algol）的语言来表达循环构造。一些Lisp黑客热爱LOOP，其他人则讨厌它。LOOP爱好者们喜欢它是因为它用了一种简洁的方式来表达特定的常用循环构造。而贬低者们不喜欢它则是因为它不太像Lisp。无论你倾向于哪一方，LOOP本身都是为语言增加新构造的宏展示其强大威力的突出示例。

7.5 DOLIST和DOTIMES

先从易于使用的DOLIST和DOTIMES宏开始。

DOLIST在一个列表的元素上循环操作，使用一个依次持有列表中所有后继元素的变量来执行循环体。[2]下面是其基本形式（去掉了一些比较难懂的选项）：

```
(dolist (var list-form)
  body-form*)
```

当循环开始时，*list-form* 被求值一次以产生一个列表。然后循环体在列表的每一项上求值一次，同时用变量*var*保存当前项的值。例如：

```
CL-USER> (dolist (x '(1 2 3)) (print x))
1
2
3
NIL
```

在这种方式下，DOLIST这种形式本身求值为NIL。

① 为了满足你的好奇心，这两个特殊操作符是TAGBODY和GO。现在无需讨论它们，第20章会介绍它们。

② DOLIST类似于Perl的foreach或Python的for。Java从Java 1.5开始作为JSR-201的一部分，增加了一个类似的增强型循环构造。注意到宏所带来的区别：一个注意到代码中常见模式的Lisp程序员可以写出一个宏来获得对该模式的源代码级抽象。一个注意到同样模式的Java程序员只能建议Sun说，这种特定的抽象值得添加到语言之中，然后Sun将会发布一个JSR并组织一个业界专家组来推敲其细节。按照Sun的说法，这一过程平均耗时18个月。在那之后，所有的编译器作者都将升级其编译器以支持新的特性，并且就算那个Java程序员所喜爱的编译器支持了这个新版本的Java，他们也仍然可能无法使用这个新特性，直到被允许打破与旧版本Java的源代码级兼容性。因此，一个Common Lisp程序员在五分钟里可以自行解决的麻烦问题却会困扰Java程序员几年时间。

如果想在列表结束之前中断一个**DOLIST**循环，则可以使用**RETURN**。

```
CL-USER> (dolist (x '(1 2 3)) (print x) (if (evenp x) (return)))
1
2
NIL
```

DOTIMES是用于循环计数的高级循环构造，其基本模板和**DOLIST**非常相似。

```
(dotimes (var count-form)
  body-form*)
```

其中的*count-form*必须要能求值为一个整数。通过每次循环，*var*所持有的整数依次为从0到比那个数小1的每一个后继整数。例如：

```
CL-USER> (dotimes (i 4) (print i))
0
1
2
3
NIL
```

和**DOLIST**一样，也可以使用**RETURN**来提前中断循环。

由于**DOLIST**和**DOTIMES**的循环体中可以包含任何类型的表达式，因此也可以使用嵌套循环。例如，为了打印出从1×1=1到20×20=400的乘法表，可以写出下面这对嵌套的**DOTIMES**循环：

```
(dotimes (x 20)
  (dotimes (y 20)
    (format t "~3d " (* (1+ x) (1+ y))))
  (format t "~%"))
```

7.6 DO

尽管**DOLIST**和**DOTIMES**方便且易于使用，却没有灵活到可用于所有循环。例如，如果想并行循环多个变量该怎样做？要是使用任意表达式来测试循环的末尾呢？如果**DOLIST**和**DOTIMES**都不能满足需求，那还可以用更通用的**DO**循环。

与**DOLIST**和**DOYIMES**只提供一个循环变量有所不同的是，**DO**允许绑定任意数量的变量，并且变量值在每次循环中的改变方式也是完全可控的也可以定义测试条件来决定何时终止循环，并可以提供一个形式，在循环结束时进行求值来为**DO**表达式整体生成一个返回值。基本模板如下所示。

```
(do (variable-definition*)
    (end-test-form result-form*)
  statement*)
```

每一个*variable-definition*引入了一个将存在于循环体作用域之内的变量。单一变量定义的完整形式是一个含有三个元素的列表。

```
(var init-form step-form)
```

上述*init-form*将在循环开始时被求值并将结果值绑定到变量*var*上。在循环的每一个后续

迭代开始之前，*step-form*将被求值并把新值分配给*var*。*step-form*是可选的，如果它没有给出，那么变量将在迭代过程中保持其值不变，除非在循环体中显式地为其赋予新值。和**LET**中的变量定义一样，如果*init-form*没有给出，那么变量将被绑定到**NIL**。另外和**LET**的情形一样的是，你可以将一个只含有名字的列表简化成一个简单的变量名来使用。

在每次迭代开始时以及所有循环变量都被指定新值后，*end-test-form*会被求值。只要其值为**NIL**，迭代过程就会继续，依次求值所有的*statement*。

当*end-test-form*求值为真时，*result-form*（结果形式）将被求值，且最后一个结果形式的值将被作为**DO**表达式的值返回。

在迭代的每一步里，所有变量的*step-form*（步长形式）将在分配任何值给变量之前被求值。这意味着可以在步长形式里引用其他任何循环变量。[①]比如在下列循环中。

```
(do ((n 0 (1+ n))
     (cur 0 next)
     (next 1 (+ cur next)))
    ((= 10 n) cur))
```

其步长形式(1+ n)、next和(+ cur next)均使用n、cur和next的旧值来求值。只有当所有步长形式都被求值以后，这些变量才被指定其新的值。（有数学天赋的读者可能会注意到，这其实是一种计算第11个斐波那契数的特别有效的方式。）

这个例子还阐述了**DO**的另一种特征——由于可以同时推进多个变量，所以往往根本不需要一个循环体。其他时候，尤其在只是把循环用作控制构造时，则可能会省略结果形式。尽管如此，这种灵活性正是**DO**表达式有点儿晦涩难懂的原因。所有这些括号都该放在哪里？理解一个**DO**表达式的最佳方式是记住其基本模板：

```
(do (variable-definition*)
    (end-test-form result-form*)
  statement*)
```

该模板中的六个括号是**DO**结构本身所必需的。一对括号来围住变量声明，一对用来围住终止测试形式和结果形式，以及一对用来围住整个表达式。**DO**中的其他形式可能需要它们自己的括号，例如变量定义总是以列表形式存在，而测试形式则通常是一个函数调用。不过**DO**循环的框架将总是一致的。下面是一些框架用黑体表示的**DO**循环的例子。

```
(do ((i 0 (1+ i)))
    ((>= i 4))
  (print i))
```

注意，该例的结果形式被省略了。不过这种用法对**DO**来说没有特别意义，因为用**DOTIMES**来写这个循环会更简单。[②]

```
(dotimes (i 4) (print i))
```

① 一个**DO**的变体**DO***，它会在求值后续变量的步长形式之前为每个变量赋值。关于它的更多细节，请查阅你喜爱的Common Lisp参考书。

② 另一个推荐使用**DOTIMES**的理由是，其宏展开将可以包含允许编译器生成更有效代码的类型声明。

另一个例子是一个没有循环体的斐波那契数计算循环：

```
(do ((n 0 (1+ n))
     (cur 0 next)
     (next 1 (+ cur next)))
    ((= 10 n) cur))
```

最后，下面循环演示了一个不绑定变量的DO循环。在当前时间小于一个全局变量值的时候，它保持循环，每分钟打印一个“Waiting”。注意，就算没有循环变量，仍然需要有那个空变量列表。

```
(do ()
    ((> (get-universal-time) *some-future-date*))
  (format t "Waiting~%")
  (sleep 60))
```

7.7　强大的 LOOP

简单的情形可以使用DOLIST和DOTIMES。但如果它们不符合需要，就需要退而使用完全通用的DO。不然还能怎样？

然而，结果是有少量的循环用法一次又一次地产生出来，例如在多种数据结构上的循环：列表、向量、哈希表和包，或是在循环时以多种方式来聚集值：收集、计数、求和、最小化和最大化。如果需要用宏来做其中的一件事（或同时几件），那么LOOP宏可以提供一种更容易表达的方式。

LOOP宏事实上有两大类——*简化的*和*扩展的*。简化的版本极其简单，就是一个不绑定任何变量的无限循环。其框架看起来像这样：

```
(loop
  body-form*)
```

主体形式在每次通过循环时都将被求值，整个循环将不停地迭代，直到使用RETURN来进行中止。例如，可以使用一个简化的LOOP来写出前面的DO循环：

```
(loop
  (when (> (get-universal-time) *some-future-date*)
    (return))
  (format t "Waiting ~%")
  (sleep 60))
```

而扩展的LOOP则是完全不同的庞然大物。值得注意的是，并非所有的Lisp程序员都喜爱扩展的LOOP语言。至少一位Common Lisp的最初设计者就很讨厌它。LOOP的贬低者们抱怨它的语法是完全非Lisp化的（换句话说，没有足够的括号）。LOOP的爱好者们则反驳说，问题在于复杂的循环构造，如果不将它们用DO那晦涩语法包装起来，它们将难于被人理解。所以他们认为最好用一种稍显冗长的语法来提供某些逻辑线索。

例如，下面是一个地道的DO循环，它将把从1到10的数字收集到一个列表中：

```
(do ((nums nil) (i 1 (1+ i)))
```

```
    ((> i 10) (nreverse nums))
  (push i nums)) → (1 2 3 4 5 6 7 8 9 10)
```

一个经验丰富的Lisp程序员将毫不费力地理解这些代码——只要理解一个DO循环的基本形式并且认识用于构建列表的PUSH/NREVERSE用法就可以了。但它并不是很直观。而它的LOOP版本理解起来就几乎可以像一个英语句子那样简单。

```
(loop for i from 1 to 10 collecting i) → (1 2 3 4 5 6 7 8 9 10)
```

接下来是一些关于LOOP简单用法的例子。下例可以对前十个平方数求和：

```
(loop for x from 1 to 10 summing (expt x 2)) → 385
```

这个用来统计一个字符串中元音字母的个数：

```
(loop for x across "the quick brown fox jumps over the lazy dog"
      counting (find x "aeiou")) → 11
```

下面这个例子用来计算第11个斐婆那契数，它类似于前面使用DO循环的版本：

```
(loop for i below 10
      and a = 0 then b
      and b = 1 then (+ b a)
      finally (return  a))
```

符号across、and、below、collecting、counting、finally、for、from、summing、then和to都是一些循环关键字，它们的存在表明当前正在使用扩展的LOOP。[①]

第22章将介绍LOOP的细节，但目前值得注意的是，我们通过它可以再次看到，宏是如何被用于扩展基本语言的。尽管LOOP提供了它自己的语言用来表达循环构造，但它并没有抹杀Lisp的其他优势。虽然循环关键字是按照循环的语法来解析的，但一个LOOP中的其余代码都是正常的Lisp代码。

另外，值得再次指出的是，尽管LOOP宏相比诸如WHEN或者UNLESS这样的宏复杂了许多，但它也只是一个宏而已。如果它没有被包括在标准库之中，你也可以自己实现它或是借助一个第三方库来实现它。

以上就是我们对基本控制构造宏的介绍。现在可以进一步了解如何定义自己的宏了。

① 循环关键字容易让人误解的一点在于它们不是关键字符号。事实上，LOOP并不关心这些符号来自什么包。当LOOP宏解析其主体时，它将等价地考察任何适当命名的符号。如果你想的话，甚至可以使用诸如:for和:across这些真正的关键字，因为它们也有正确的名字。但多数人只用普通符号。由于循环关键字仅被用作句法标记，因此将它们用做其他目的，如作为函数或变量的名字，也是没有关系的。