第8章

如何自定义宏

现在可以开始编写自己的宏了。前一章里提及的标准宏暗示了可以用宏做到的某些事情，但这只是开始。

相比于C语言的函数可以让每个C程序员编写C标准库中的函数的简单变体，Common Lisp的宏也无非是可以让每个Lisp程序员创建他们自己的标准控制构造变体罢了。作为语言的一部分，宏能够用于在核心语言和标准库之上创建抽象，从而使你更直接地表达想表达的事物。

具有讽刺意义的是，也许对于宏的正确理解，最大的障碍是它们已经很好地集成到了语言里。在许多方面，它们看起来只是一些有趣的函数——它们用Lisp写成，接受参数并返回结果。同时，它们允许你抽象那些分散注意力的细节。尽管有这些相似性，但宏的操作层面与函数不同，而且它还有着完全不同类型的抽象。

一旦理解了宏与函数之间的区别，你就会发现这门语言中宏的紧密集成所带来的巨大优势。但同时它也是经常导致新程序员困惑的主要原因。下面来讲个故事，尽管从历史或技术意义上来说并不真实，然而通过这种方式，你倒是可以思考一下宏的工作方式，以此来缓解一下困惑。

8.1 Mac的故事：只是一个故事

很久以前，有一个由Lisp程序员们所组成的公司。那个年代相当久远，所以Lisp还没有宏。每次，任何不能用函数来定义或是用特殊操作符来完成的事情都不得不完全通过手写来实现，这带来了很大的不便。不幸的是，这个公司的程序员们虽然杰出却非常懒惰。在他们的程序中，当需要编写大量单调乏味的代码时，他们往往会写下一个注释来描述想要在该位置上编写的代码。更不幸的是，由于很懒惰，他们也很讨厌回过头去实际编写那些注释所描述的代码。不久，这个公司就有了一大堆无法运行的程序，因为它们全都是代表着尚需编写代码的注释。

走投无路之下，老板雇用了一个初级程序员Mac。他的工作就是找到这些注释，编写所需的代码，然后再用其替换掉程序中的注释。Mac从未运行过这些程序——程序尚未完成，他当然运行不了。但就算这些程序完成了，Mac也不知道该用怎样的输入来运行它们。因此，他只是基于注释的内容来编写他的代码，再将其发还给最初的程序员。

在Mac的帮助下，不久之后，所有的程序都完成了，公司通过销售它们赚了很多钱，并用这些钱将其程序员团队扩大了一倍。但不知为何，没有人想到要雇用其他人来帮助Mac。很快他就

开始单枪匹马地同时协助几十个程序员了。为了避免将他所有的时间都花在搜索源代码的注释上，Mac对程序员们使用的编译器做了一个小小的更改。从那以后，只要编译器遇到一个注释，它就会将注释以电子邮件的形式发给他并等待他将替换的代码传送回来。然而，就算有了这个变化，Mac也很难跟上程序员的进度。他尽可能小心地工作，但有时，尤其是当注释不够清楚时，他会犯错误。

不过程序员们注意到了，他们将注释写得越精确，Mac就越有可能发回正确的代码。一天，一个花费大量时间用文字来描述他想要的代码的程序员，在他的注释里写入了一个可以生成他想要的代码的Lisp程序。这对Mac来说很简单，他只需运行这个程序并将结果发给编译器就好了。

接下来又出现了一种创新。有一个程序员在他程序的开始处写了一段备注，其中含有一个函数定义以及另一个注释，该注释为："Mac，不要在这里写任何代码，但要把这个函数留给以后使用，我将在我的其他一些注释里用到它。"同一个程序里还有如下的注释："Mac，将这个注释替换成用符号x和y作为参数来运行上面提到的那个函数所得到的结果。"

这项技术在几天里就迅速流行起来，多数程序都含有数十个注释，它们定义了那些只被其他注释中的代码所使用的函数。为了使Mac更容易地辨别那些只含有定义而不必立即回复的注释，程序员们用一个标准前缀来标记它们："给Mac的定义，仅供阅读。"（Definition for Mac, Read Only.）由于程序员们仍然很懒惰，这个写法很快简化成"DEF. MAC. R/O"，接着又被简化为"DEFMACRO"。

不久以后，这些给Mac的注释中再没有实际可读的英语了。Mac每天要做的事情就是阅读并反馈那些来自编译器的含有DEFMACRO注释的电子邮件，以及调用那些DEFMACRO里所定义的函数。由于注释中的Lisp程序做了所有实际的工作，跟上这些电子邮件的进度完全没有问题。Mac手头上突然有了大量时间，可以坐在他的办公室里做那些关于白色沙滩、蓝色海水和鸡尾酒的白日梦了。

几个月以后，程序员们意识到已经很长时间没人见过Mac了。当他们去他的办公室时，发现所有东西上都积了薄薄的一层灰，一个桌子上还放着几本热带地区的旅行手册，而电脑则是关着的。但是编译器仍在正常工作——这怎么可能？看起来Mac对编译器做了最后一个修改：现在不需要用电子邮件将注释发给Mac了，编译器会将那些DEFMACRO中所定义的函数保存下来，并在其被其他注释调用时运行它们。程序员们觉得没有理由告诉老板Mac不再来办公室了。因此直到今天，Mac还领着薪水，并且时不时地会从某个热带地区给程序员们发一张明信片。

8.2 宏展开期和运行期

理解宏的关键在于必须清楚地知道那些生成代码的代码（宏）和那些最终构成程序的代码（所有其他内容）之间的区别。当编写宏时，你是在编写那些将被编译器用来生成代码并随后编译的程序。只有当所有的宏都被完全展开并且产生的代码被编译后，程序才可以实际运行。宏运行的时期被称为宏展开期（macro expansion time），这和运行期（runtime）是不同的，后者是正常的代码（包括那些由宏生成的代码）实际运行的阶段。

牢记这一区别很重要，因为运行在宏展开期的代码与那些运行在运行期的代码相比，它们的运行环境完全不同。也就是说，在宏展开期无法访问那些仅存在于运行期的数据。正如Mac无法

运行他写的程序是因为不知道正确的输入那样，运行在宏展开期的代码也只能处理那些来自源代码本身的数据。例如，假设在程序的某个地方出现了下面这样的源代码：

```
(defun foo (x)
  (when (> x 10) (print 'big)))
```

正常情况下，你将x设为一个变量，用它保存传递给一个对foo调用的实参。但在宏展开期，比如当编译器正在运行**WHEN**宏的时候，唯一可用的数据就是源代码。由于程序尚未运行，没有对foo的调用，因此也没有值关联到x上。相反，编译器传递给**WHEN**的值只是代表源代码的Lisp列表，也即(> x 10)以及(print 'big)。假设**WHEN**确如前一章中所见的那样用类似下面的宏定义而成。

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

那么当foo中的代码被编译时，**WHEN**宏将以那两个形式作为实参来运行。形参condition会被绑定到形式(> x 10)上，而形式(print 'big)会被收集到一个列表中成为**&rest** body形参的值。那个反引用表达式将随后通过插入condition的值，并将body的值嵌入**PROGN**的主体来生成下面的代码：

```
(if (> x 10) (progn (print 'big)))
```

当Lisp被解释而非编译时，宏展开期和运行期之间的区别不甚明显，因为它们临时纠缠在了一起。同样，语言标准并未规定解释器处理宏的具体方式——它可能在被解释的形式中展开所有的宏，然后解释执行那些宏所生成的代码，也可能是直接解释一个形式并在每次遇到宏的时候才展开。无论哪种情况，总是向宏传递那些代表宏形式中子形式的未经求值的Lisp对象，并且宏的作用仍然是生成做某些事情的代码，而非直接做任何事情。

8.3 DEFMACRO

如同你在第3章里所看到的那样，宏真的是用**DEFMACRO**来定义的。当然，它代表的是“定义宏”（DEFine MACRO）而不是“给Mac的定义”（Definition for Mac）。**DEFMACRO**的基本框架和**DEFUN**框架很相似。

```
(defmacro name (parameter*)
  "Optional documentation string."
  body-form*)
```

和函数一样，宏由名字、形参列表、可选文档字符串以及Lisp表达式体所构成。[①]但如前所述，宏并不是直接做事，它只是用于生成以后工作所需的代码。

宏可以使用Lisp的所有功能来生成其展开式，这意味着本章只能初步说明宏的具体功用。不过我可以描述一个通用的宏编写过程，它适用于从最简单到最复杂的所有宏。

宏的工作是将宏形式（首元素为宏名的Lisp形式）转化成做特定事情的代码。有时是从想要

① 和函数一样，宏也可以含有声明，但你现在不需要考虑它们。

编写的代码开始来编写宏的，就是说从一个示例的宏形式开始。其他时候则是在连续几次编写了相同的代码模式并认识到通过抽象该模式可以使代码更清晰后，才开始决定编写宏的。

无论从哪一端开始，你都需要在开始编写宏之前搞清楚另一端：既需要知道从哪里开始，又要知道正在向何处去，然后才能期待编写代码来自动地做到这点。因此编写宏的第一步是至少应去编写一个宏调用的示例以及该调用应当展开成的代码。

一旦有了示例调用及预想的展开式，那么就可以开始第二步了：编写实际的宏代码。对于简单的宏来说，这将极其轻松——编写一个反引用模板并将宏参数插入到正确的位置上。复杂的宏则会是一个庞大的独立程序，它将带有配套的助手函数和数据结构。

在已经编写了代码来完成从示例调用到适当的展开式的转换以后，需要确保宏所提供的抽象没有“泄漏”其实现细节。有漏洞的宏抽象将只适用于特定参数上，或会以预想之外的方式与调用环境中的代码进行交互。后面将会看到，宏只能以很少的几种方式泄漏，而所有这些都是可以轻易避免的，只要知道如何检查它们就行。8.7节将讨论具体的方法。

总结起来，编写宏的步骤如下所示：

(1) 编写示例的宏调用以及它应当展开成的代码，反之亦然；

(2) 编写从示例调用的参数中生成手写展开式的代码；

(3) 确保宏抽象不产生“泄漏”。

8.4 示例宏：do-primes

为了观察这三步过程是怎样发挥作用的，下面将编写一个宏do-primes，它提供了一个类似DOTIMES和DOLIST的循环构造，只是它并非迭代在整数或者一个列表的元素上，而是迭代在相继的素数上。这并不是一个特别有用的宏，它只是在演示该过程。

首先你需要两个工具函数：一个用来测试给定的数是否为素数，另一个用来返回大于或等于其实参的下一个素数。这两种情况都可以使用简单而低效的暴力手法来解决。

```
(defun primep (number)
  (when (> number 1)
    (loop for fac from 2 to (isqrt number) never (zerop (mod number fac)))))

(defun next-prime (number)
  (loop for n from number when (primep n) return n))
```

现在就可以写这个宏了。按照前面所概括的过程，至少需要一个宏调用示例以及它应当展开成的代码。假设你开始时想通过如下代码来表示循环：

```
(do-primes (p 0 19)
  (format t "~d " p))
```

这个循环在每个大于等于0并小于等于19的素数上分别依次执行循环体，并以变量p保存当前素数。仿照标准的DOLIST和DOTIMES宏来定义是合理的。按照已有宏的模式操作的宏比那些引入了无谓的新颖语法的宏更易于理解和使用。

如果没有do-primes宏，你可以用DO（和前面定义的两个工具函数）来写出下面这个

循环：

```
(do ((p (next-prime 0) (next-prime (1+ p))))
    ((> p 19))
  (format t "~d " p))
```

现在就可以开始编写将前者转化成后者的代码了。

8.5 宏形参

由于传递给宏的实参是代表宏调用源代码的Lisp对象，因此任何宏的第一步工作都是提取出那些对象中用于计算展开式的部分。对于那些简单地将其实参直接插入到模板中的宏而言，这一步骤相当简单：只需定义正确的形参来保存不同的实参就可以了。

但是这一方法似乎并不适用于do-primes。do-primes调用的第一个参数是一个列表，其含有循环变量的名字p及其下界0和上界19。但如果查看展开式就会发现，该列表作为整体并没有出现在展开式中，三个元素被拆分开并分别放在不同的位置上。

可以用两个形参来定义do-primes，一个用来保存该列表，另一个**&rest**形参保存形式体，然后手工分拆该列表，类似下面这样：

```
(defmacro do-primes (var-and-range &rest body)
  (let ((var (first var-and-range))
        (start (second var-and-range))
        (end (third var-and-range)))
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
         ((> ,var ,end))
       ,@body)))
```

稍后，将解释上述宏形式体怎样生成正确的展开式。目前只需注意变量var、start和end都持有一个从var-and-range中提取出的值，它们随后被插入到反引用表达式中以生成do-primes的展开式。

尽管如此，但并不需要“手工”分拆var-and-range，因为宏形参列表是所谓的解构（destructuring）形参列表。顾名思义，“解构”涉及分拆一个结构体，在本例中是传递给一个宏的列表结构形式。

在解构形参列表中，简单的形参名将被替换成嵌套的形参列表。嵌套形参列表中的形参将从绑定到该形参列表的表达式的元素中获得其值。例如可以将var-and-range替换成一个列表(var start end)，然后这个列表的三个元素将被自动解构到三个形参上。

宏形参列表的另一个特性是可以使用**&body**作为**&rest**的同义词。**&body**和**&rest**在语义上是等价的，但许多开发环境根据一个**&body**形参的存在来修改它们缩进那些使用该宏的代码的方式。通常，**&body**被用来保存一个构成该宏主体的形式的列表。

因此你可以通过将do-primes定义成下面这样来完成其定义，并同时向读者和你的开发工具说明它的用途：

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
```

```
      ((> ,var ,end))
    ,@body))
```

除了更加简洁以外，解构形参列表还可以让你自动检查错误。通过以这种方式定义do-primes，Lisp可以检测到那些首参数不是三元素列表的调用，并提供有意义的错误信息，就好像你用太多或太少的参数调用了一个函数那样。同样，在诸如SLIME这样的开发环境中，只要输入一个函数或宏的名字就可以指示它所期待的参数。如果你使用了一个解构形参列表，那么环境将可以更明确地告诉你宏调用的语法。使用最初的定义，SLIME将告诉你do-primes可以像这样来调用：

```
(do-primes var-and-range &rest body)
```

但在新定义下，它可以告诉你一个调用应当看起来像这样：

```
(do-primes (var start end) &body body)
```

解构形参列表可以含有**&optional**、**&key**和**&rest**形参，并且可以含有嵌套的解构列表。尽管如此，你在编写do-primes的过程中却不需要任何这些选项。

8.6 生成展开式

由于do-primes是一个相当简单的宏，在解构了参数以后，剩下的就是将它们插入到一个模板中来得到展开式。

对于像do-primes这样简单的宏，反引用语法刚好合适。回顾一下，反引用表达式与引用表达式很相似，除了可以“解引用”（unquote）特定的值表达式，即前面加上逗号，可能其后还会接着一个“@”符号。没有这个“@”符号，逗号会导致子表达式的值被原样包含。有了“@”符号，其值（必须是一个列表）可被“拼接”到其所在的列表中。

采用另一种方式也会有助于理解反引用语法，这就是将其视为编写生成列表的代码的一种特别简洁的方式。这种理解方式的优点是可以相当明确地看到其表象之下实际发生的事——当读取器读到一个反引用表达式时，它将其翻译成生成适当列表结构的代码。例如，`(,a b)可以被读取成(list a 'b)。语言标准并未明确指定读取器必须产生怎样的代码，只要它能生成正确的列表结构就可以了。

表8-1给出了一些反引用表达式的范例，同时带有与之等价的列表构造代码以及其中任意一种形式的求值结果。[1]

表8-1 反引用表达式的例子

反引用语法	等价的列表构造代码	结 果
`(a (+ 1 2) c)	(list 'a '(+ 1 2) 'c)	(a (+ 1 2) c)
`(a ,(+ 1 2) c)	(list 'a (+ 1 2) 'c)	(a 3 c)
`(a (list 1 2) c)	(list 'a '(list 1 2) 'c)	(a (list 1 2) c)
`(a ,(list 1 2) c)	(list 'a (list 1 2) 'c)	(a (1 2) c)
`(a ,@(list 1 2) c)	(append (list 'a) (list 1 2) (list 'c))	(a 1 2 c)

① APPEND函数尚未提及，它能够接受任意数量的列表实参并返回一个由它们拼接而成的单独列表。

重要的是要注意反引用只是一种便利措施。不过这确实相当便利。为了说明究竟怎么便利，我们可以将do-primes的反引用版本和下面的版本作比较，后者使用了显式的列表构造代码：

```
(defmacro do-primes-a ((var start end) &body body)
  (append '(do)
          (list  (list (list var
                             (list 'next-prime start)
                             (list 'next-prime (list '1+ var)))))
          (list (list (list '> var end)))
          body))
```

稍后即将看到，do-primes的当前实现并不能正确地处理特定的临界情况，但首先应当确认它至少应能适用于最初的例子。可以用两种方式来测试它。可以简单地通过使用来间接地测试，也就是说，如果结果的行为是正确的，那么展开式很可能就是正确的。例如，可以将do-primes最初的用例键入到REPL中，你会看到它确实打印出了正确的素数序列。

```
CL-USER> (do-primes (p 0 19) (format t "~d " p))
2 3 5 7 11 13 17 19
NIL
```

或者也可以通过查看特定调用的展开式来直接检查该宏。函数**MACROEXPAND-1**接受任何Lisp表达式作为参数并返回做宏展开一层的结果。[①]由于**MACROEXPAND-1**是一个函数，所以为了传给它一个字面的宏形式，就必须引用它。可以用它来查看前面调用的展开式。[②]

```
CL-USER> (macroexpand-1 '(do-primes (p 0 19) (format t "~d " p)))
(DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
    ((> P 19))
  (FORMAT T "~d " P))
T
```

或者，在SLIME中也可以更方便地检查一个宏的展开式：将光标放置在源代码中一个宏形式的开括号上，并输入C-c RET来调用Emacs函数slime-macroexpand-1，后者将把宏调用传递到**MACROEXPAND-1**上，并将结果“美化输出”到一个临时缓冲区上。

无论怎样得到展开式，你都可以看到宏展开的结果和最初的手写展开式是一样的，因此看起来do-primes是有效的。

8.7　堵住漏洞

Jeol Spolsky在他的随笔“The Law of Leaky Abstractions”里创造了术语“有漏洞的抽象”(leaky abstraction)，以此来描述一种抽象：“泄露”了本该抽象的细节。由于编写宏是一种创造抽象的

① 另一个函数**MACROEXPAND**将持续地展开结果，只要返回的展开式的第一个元素是宏的名字，它就会不断进行下去。但它通常会深入展示代码行为，其程度往往远超出你所预期。因为诸如DO这类基本控制构造也被实现为宏。换句话说，尽管它对看到宏最终可展开成怎样的代码具有一定教育意义，但这对于了解宏的作用并不是一个有用的视角。

② 如果所有宏展开被显示在一行里，这很有可能是因为变量***PRINT-PRETTY***为**NIL**。如果是这样，求值(setf *print-pretty* t)将使展开式更易于阅读。

方式，故此需要确保宏不产生不必要的泄露。[1]

如同即将看到的，宏可能以三种方式泄露其内部工作细节。幸运的是，你可以相当容易地看出一个给定的宏是否存在着任何一种泄露方式，并修复它。

当前的宏定义存在三种可能的宏泄露中的一种，确切地说，它会过多地对end子形式求值。假设没有使用诸如19这样的字面数字，而是用像(random 100)这样的表达式在end的位置上来调用do-primes：

```
(do-primes (p 0 (random 100))
  (format t "~d " p))
```

假设这里的意图是要在从0到由(random 100)所返回的任意随机数字的范围内循环查找素数。但**MACROEXPAND-1**的结果显示这并不是当前实现所做的事。

```
CL-USER> (macroexpand-1 '(do-primes (p 0 (random 100)) (format t "~d " p)))
(DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
    ((> P (RANDOM 100)))
  (FORMAT T "~d " P))
T
```

当我们运行展开式代码时，**RANDOM**将在每次进行循环的终止测试时被求值一次。这样，循环将不会在p大于一个初始给定的随机数时终止，而是在循环刚好生成一个小于或等于当前p值的随机数时，循环才会终止。由于循环的整体次数仍然是随机的，因此它将产生一个与**RANDOM**所返回的统一分布相当不同的分布形式。

这就是一种抽象中的漏洞，因为为了正确使用该宏，调用者必须注意end形式被求值超过一次的情况。一种堵住漏洞的方式是简单地将其定义成do-primes的行为。但这并不非常令人满意，你在实现宏时应当试图遵守最少惊动原则（Principle of Least Astonishment）。而且通常情况下，程序员们希望他们传递给宏的形式除非必要将不会被多次求值。[2]更进一步，由于do-primes是构建在标准宏**DOTIMES**和**DOLIST**之上的，而这两个宏都不会导致其循环体之外的形式被多次求值，所以多数程序员将期待do-primes具有相似的行为。

修复多重求值问题是相当容易的：只需生成代码来对end求值一次，并将其值保存在一个稍后将会用到的变量里。回想在**DO**循环中，用一个初始形式但没有步长形式来定义的变量并不会在迭代过程中改变其值。因此可以用下列定义来修复多重求值问题：

```
(defmacro do-primes ((var start end) &body body)
  `(do ((ending-value ,end)
        (,var (next-prime ,start) (next-prime (1+ ,var))))
       ((> ,var ending-value))
     ,@body))
```

① 该随笔出自Joel Spolsky的*Joel on Software*，你也可以查阅http://www.joelonsoftware.com/articles/LeakAbstractions.html来获取相关内容。Spolsky在随笔中的观点是，所有的抽象都在某种意义上存在泄露。也就是说，不存在完美的解决方案。但这也不意味着你可以容忍那些可以轻易堵上的漏洞。

② 当然，特定形式其本意就是被多次求值，例如一个do-primes循环体中的形式。

然而不幸的是，这一修复却又给宏抽象引入了两个新漏洞。

其中一个新漏洞类似于刚修复的多重求值漏洞。因为在DO循环中，变量的初始形式是以变量被定义的顺序来求值的，当宏展开被求值时，传递给end的表达式将在传递给start的表达式之前求值，这与它们出现在宏调用中的顺序相反。并在start和end都是像0和19这样的字面值时，这一泄露，不会带来任何问题。但当它们是可以产生副作用的形式时，不同的求值顺序将使它们再次违反最少惊动原则。

通过交换两个变量的定义顺序就可轻易堵上该漏洞。

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
        (ending-value ,end))
       ((> ,var ending-value))
     ,@body))
```

最后一个需要堵上的漏洞是由于使用了变量名ending-value而产生的。问题在于这个名字（其应当完全属于宏实现内部的细节）它可以跟传递给宏的代码或是宏被调用的上下文产生交互。下面这个看似无辜的do-primes调用会由于这个漏洞而无法正常工作。

```
(do-primes (ending-value 0 10)
  (print ending-value))
```

这样也不可以。

```
(let ((ending-value 0))
  (do-primes (p 0 10)
    (incf ending-value p))
  ending-value)
```

MACROEXPAND-1再次向你展示问题所在。第一次调用展开成这样。

```
(do ((ending-value (next-prime 0) (next-prime (1+ ending-value)))
     (ending-value 10))
    ((> ending-value ending-value))
  (print ending-value))
```

某些Lisp可能因为ending-value作为变量名在同一个DO循环中被用了两次而拒绝上面的代码。如果没有被完全拒绝，上述代码也将无限循环下去，由于ending-value永远不会大于其自身。

第二个问题调用展开成下面的代码：

```
(let ((ending-value 0))
  (do ((p (next-prime 0) (next-prime (1+ p)))
       (ending-value 10))
      ((> p ending-value))
    (incf ending-value p))
  ending-value)
```

在这种情况下生成的代码是完全合法的，但其行为完全不是你想要的那样。由于在循环之外由**LET**所建立的ending-value绑定被DO内部同名的变量所掩盖，形式(incf ending-valuep)

将递增循环变量ending-value而不是同名的外层变量，因此得到了另一个无限循环。[1]

很明显，为了补上这个漏洞，需要一个永远不会在宏展开代码之外被用到的符号。可以尝试使用一个真正罕用的名字，但即便如此也不可能做到万无一失。也可以使用第21章里介绍的包(package)，从而在某种意义上起到保护作用。但还有一个更好的解决方案。

函数**GENSYM**在其每次被调用时返回唯一的符号。这是一个没有被Lisp读取器读过的符号并且永远不会被读到，因为它不会进入到任何包里。因而就可以在每次do-primes被展开时生成一个新的符号以替代像ending-value这样的字面名称。

```
(defmacro do-primes ((var start end) &body body)
  (let ((ending-value-name (gensym)))
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
          (,ending-value-name ,end))
         ((> ,var ,ending-value-name))
       ,@body)))
```

注意调用**GENSYM**的代码并不是展开式的一部分，它作为宏展开器的一部分来运行从而在每次宏被展开时创建一个新符号。这初看起来有一点奇怪——ending-value-name是一个变量，其值是另一个变量名。但其实它和值为一个变量名的形参var并没有什么区别，区别在于var的值是由读取器在宏调用被读取时创建的，而ending-value-name的值则是在宏代码运行时由程序化生成的。

使用这个定义，前面两个有问题的形式现在就可以展开成按预想方式运作的代码了。第一个形式

```
(do-primes (ending-value 0 10)
  (print ending-value))
```

展开成下面的代码：

```
(do ((ending-value (next-prime 0) (next-prime (1+ ending-value)))
     (#:g2141 10))
    ((> ending-value #:g2141))
  (print ending-value))
```

现在用来保存循环终值的变量是生成符号，#:g2141。该符号的名字G2141是由**GENSYM**所生成的，但这并不重要，重要的是这个符号的对象标识。生成符号是以未保留符号通常的语法形式打印出来的，带有前缀#:。

另一个之前有问题的形式：

```
(let ((ending-value 0))
  (do-primes (p 0 10)
    (incf ending-value p))
  ending-value)
```

① 该循环在给定任意素数下的无限性并非是显而易见的。为了证明其确实是无限的，起始点是Bertrand公设：对任何n>1都存在一个素数p，n<p<2n。由此你就可以证明，对于给定的任意素数，P总是小于它之前的所有素数之和，而下一个素数P'也同样小于前面的这个和再加上P。

如果将do-primes形式替换成其展开式的话，以上形式将会变成这样：

```
(let ((ending-value 0))
  (do ((p (next-prime 0) (next-prime (1+ p)))
       (#:g2140 10))
      ((> p #:g2140))
    (incf ending-value p))
  ending-value)
```

再一次，由于do-primes循环外围的**LET**所绑定的变量ending-value不再被任何由展开代码引入的变量所掩盖，因此再没有漏洞了。

并非宏展开式中用到的所有字面名称都会导致问题。等你对于多种绑定形式有了更多经验以后，将可以鉴别一个用在某个位置上的给定名字是否会导致在宏抽象中出现漏洞。但安全起见，使用一个符号生成的名字并没有什么坏处。

利用这些修复就可以堵上do-primes实现中的所有漏洞了。一旦积累了一点宏编写方面的经验以后，你将获得在预先堵上这几类漏洞的情况下编写宏的本领。事实上做到这点很容易，只须遵循下面所概括的这些规则即可。

- 除非有特殊理由，否则需要将展开式中的任何子形式放在一个位置上，使其求值顺序与宏调用的子形式相同。
- 除非有特殊理由，否则需要确保子形式仅被求值一次，方法是在展开式中创建变量来持有求值参数形式所得到的值，然后在展开式中所有需要用到该值的地方使用这个变量。
- 在宏展开期使用GENSYM来创建展开式中用到的变量名。

8.8　用于编写宏的宏

当然，没有理由表明只有在编写函数的时候才能利用宏的优势。宏的作用是将常见的句法模式抽象掉，而反复出现在宏的编写中的特定模式同样也可受益于其抽象能力。

事实上，你已经见过了这样一种模式。许多宏，例如最后版本的do-primes，它们都以一个LET形式开始，后者引入了一些变量用来保存宏展开过程中用到的生成符号。由于这也是一个常见模式，那为什么不用一个宏来将其抽象掉呢？

本节将编写一个宏with-gensyms，它刚好做到这点。换句话说，你将编写一个用来编写宏的宏：一个宏用来生成代码，其代码又生成另外的代码。尽管在你习惯于在头脑中牢记不同层次的代码之前，可能会对复杂的编写宏的宏感到有一点困惑，但with-gensyms是相当简单的，而且还可以当作一个有用但又不会过于浪费脑筋的练习。

所写的宏应类似于下面这种形式。

```
defmacro do-primes ((var start end) &body body)
  (with-gensyms (ending-value-name)
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
          (,ending-value-name ,end))
         ((> ,var ,ending-value-name))
       ,@body)))
```

并且还需要让其等价于之前版本的do-primes。换句话说，with-gensyms需要展开成一个**LET**，它会把每一个命名的变量（在本例中是ending-value-name）都绑定到一个生成符号上。很容易就可以写出一个简单的反引用模板。

```
(defmacro with-gensyms ((&rest names) &body body)
  `(let ,(loop for n in names collect `(,n (gensym)))
     ,@body))
```

注意你是怎样用一个逗号来插入**LOOP**表达式的值的。这个循环生成了一个绑定形式的列表，其中每个绑定形式由一个含有with-gensyms中的一个给定名字和字面代码(gensym)的列表所构成。你可以通过将name替换成一个符号的列表，从而在REPL中测试**LOOP**表达式生成的代码。

```
CL-USER> (loop for n in '(a b c) collect `(,n (gensym)))
((A (GENSYM)) (B (GENSYM)) (C (GENSYM)))
```

在绑定形式的列表之后，with-gensyms的主体参数被嵌入到**LET**的主体之中。这样，被封装在一个with-gensyms中的代码将可以引用任何传递给with-gensyms的变量列表中所命名的变量。

如果在新的do-primes定义中对with-gensyms形式进行宏展开，就将看到下面这样的结果：

```
(let ((ending-value-name (gensym)))
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
        (,ending-value-name ,end))
       ((> ,var ,ending-value-name))
     ,@body))
```

看起来不错。尽管这个宏相对简单，但重要的是要清楚地了解不同的宏是分别在何时被展开的：当编译关于do-primes的**DEFMACRO**时，with-gensyms形式就被展开成刚刚看到的代码并被编译了。这样，do-primes的编译版本就已经跟你手写外层的**LET**时一样了。当编译一个使用了do-primes的函数时，由with-gensyms生成的代码将会运行用来生成do-primes的展开式，但with-gensyms宏本身在编译一个do-primes形式时并不会被用到，因为在do-primes被编译时，它早已经被展开了。

另一个经典的用于编写宏的宏：once-only

另一个经典的用于编写宏的宏是once-only，它用来生成以特定顺序仅求值特定宏参数一次的代码。使用once-only，你几乎可以跟最初的有漏洞版本一样简单地写出do-primes来，就像这样：

```
(defmacro do-primes ((var start end) &body body)
  (once-only (start end)
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
         ((> ,var ,end))
       ,@body)))
```

尽管如此，但如果详加解释的话，once-only的实现将远远超出本章的内容，因为它依赖于多层的反引用和解引用。如果真想进一步提高宏技术的话，你可以尝试分析它的工作方式。如下所示：

```
(defmacro once-only ((&rest names) &body body)
  (let ((gensyms (loop for n in names collect (gensym))))
    `(let (,@(loop for g in gensyms collect `(,g (gensym))))
       `(let (,,@(loop for g in gensyms for n in names collect ``(,,g ,,n)))
          ,(let (,@(loop for n in names for g in gensyms collect `(,n ,g)))
             ,@body)))))
```

8.9　超越简单宏

当然，我可以说更多关于宏的事情。目前为止，所有你见到的宏都是相当简单的例子，它们帮助你减轻了一些写代码的工作量，但却并没有提供表达事物的根本性的新方式。在接下来的章节里你将看到一些宏的示例，它们允许你以一种假如没有宏就完全做不到的方式来表达事物。从下一章开始，你将构建一个简单而高效的单元测试框架。