第20章 特殊操作符 20

从某种意义上说，对于前一章里所描述的状况系统，其最令人印象深刻的方面在于，如果它还不是语言的一部分，那么它完全可以被实现成用户层面的库。这种可能性体现在，尽管没有哪个Common Lisp的特殊操作符是直接用于产生或处理状况的，但它们提供了通向语言底层机制的足够的权限，从而能够做到诸如控制栈的回退这样的事情。

我在前面的章节里已经讨论了大多数常用的特殊操作符，但有两个理由使我们有必要熟悉其他的特殊操作符。首先，一些不太常用的特殊操作符之所以不太常用，只是因为需要用它们来处理的情况也很少发生。你有必要熟悉这些特殊操作符，这样当有一天需要它们时，你至少可以想到它们。其次，因为25个特殊操作符（连同求值函数调用的基本规则以及内置数据类型）提供了语言其余部分的基础，因此对它们有整体的了解将有助于你理解该语言的工作方式。

在本章里，我将讨论所有的特殊操作符，其中的一些只是简要介绍，另一些则会详细叙述，你会看到它们是如何在一起工作的。我将指出它们中的哪些是可以直接用在你自己的代码中的，哪些是用来作为你一直使用的其他控制构造的基础的，以及哪些是你将很少直接使用但在宏生成的代码中却是相当有用的。

20.1 控制求值

第一类特殊操作符包括三个对形式求值提供基本控制的操作符，它们是QUOTE、IF和PROGN，我们已经全部讨论过了。尽管如此，还是该注意这些操作符是如何分别对一个或多个形式求值提供基本控制类型的。QUOTE完全避免求值，从而允许你得到作为数据的S-表达式。IF提供了基本的布尔选择操作符，从而构造出其他所有的条件执行构造。[1] PROGN则提供了序列化一组形式的能力。

20.2 维护词法环境

特殊操作符中最大的一类由那些维护和访问词法环境（lexical environment）的操作符所组成。

① 当然，如果IF不是一个特殊操作符而其他某个条件形式（例如COND）是的话，那么你也可以将IF构造成一个宏。事实上，在许多Lisp方言里，从McCarthy最初的Lisp开始，COND都是那个最基本的条件求值操作符。

前面已经讨论的LET和LET*就是用于维护词法环境的特殊操作符，因为它们可以引入新的词法变量绑定。任何诸如DO或DOTIMES这类绑定了词法变量的构造都将被展开成一个LET或LET*[①]。SETQ特殊操作符是一种用来访问词法环境的特殊操作符，因为可以用它设置那些由LET和LET*所创建的绑定。

不过，变量并不是唯一可以在词法作用域里命名的东西。虽然大多数函数都是通过DEFUN全局定义的，但也可能通过特殊操作符FLET和LABELS创建局部函数，通过MACROLET创建局部宏，以及通过SYMBOL-MACROLET创建一种特殊类型的宏，称为符号宏（symbol macro）。

LET允许你引入一个词法变量，其作用域是LET形式的主体；同样，FLET和LABEL可以让你定义一个函数，使其只能在FLET或LABELS形式的作用域内被访问。在你需要一个比内联定义的LAMBDA表达式更复杂的局部函数，或是将多次使用的局部函数时，这些特殊操作符将会非常有用。两者具有相同的基本形式，看起来像下面这样：

```
(flet (function-definition*)
  body-form*)
```

以及

```
(labels (function-definition*)
  body-form*)
```

其中每个*function-definition*具有下面的形式：

```
(name (parameter*) form*)
```

FLET与LABELS之间的区别在于，由FLET所定义的函数名只能在FLET的主体中使用，而由LABELS所引入的名字却可以立即使用，包括LABELS所定义的函数本身。这样，LABELS可以用来定义递归函数，而FLET就不行。FLET不能用来定义递归函数虽然看起来是一种限制，但Common Lisp同时提供了FLET和LABELS，这是因为有时能够编写出一个调用另一个同名函数的局部函数也是有用的，被调用的同名函数可能是一个全局定义的函数或是来自外围作用域的另一个局部函数。

在一个FLET或LABELS的主体内，你可以像任何其他函数那样使用这些局部函数的名字，包括使用FUNCTION特殊操作符。由于可以使用FUNCTION来获得代表FLET或LABELS所定义函数的函数对象，并且因为一个FLET或LABELS可以定义在其他诸如LET这样的绑定形式的作用域内，所以这些函数可能是闭包。

因为局部函数可以引用来自其外围作用域的变量，所以它们通常可以书写成比等价的辅助函数带有更少参数的形式。这在你需要传递一个只接受单一参数作为函数参数的函数时尤为方便。例如，在下面的函数中（你在第25章里还会再次看到它），FLET所定义的函数count-version接受单一参数，这是walk-directory所要求的，但该函数还用到了由外围LET所引入的变量versions：

① 从技术上来讲，这些构造也可以展开成一个LAMBDA表达式，正如我在第6章里所提到的，因为LET可以被定义成一个展开成对匿名函数调用的宏，而在一些早期的Lisp实现里确实就是这样做的。

```
(defun count-versions (dir)
  (let ((versions (mapcar #'(lambda (x) (cons x 0)) '(2 3 4))))
    (flet ((count-version (file)
             (incf (cdr (assoc (major-version (read-id3 file)) versions)))))
      (walk-directory dir #'count-version :test #'mp3-p))
    versions))
```

这个函数也可以写成在**FLET**定义的count-version位置上使用一个匿名函数，给这个函数一个有意义的名字更易于阅读它。

另外，当一个辅助函数需要进行递归时，使用匿名函数不可能做到这点。[1]当你不想把一个递归的辅助函数定义成全局函数时，可以使用**LABELS**。例如，下面的函数collect-leaves使用递归的辅助函数walk来遍历一棵树，并把树中所有的原子收集到一个列表里，该列表（在求逆以后）由collect-leaves返回：

```
(defun collect-leaves (tree)
  (let ((leaves ()))
    (labels ((walk (tree)
               (cond
                 ((null tree))
                 ((atom tree) (push tree leaves))
                 (t (walk (car tree))
                    (walk (cdr tree))))))
      (walk tree))
    (nreverse leaves)))
```

要再次注意，在walk函数里可以引用变量leaves，它是由外围的**LET**引入的。

FLET和**LABELS**在用于宏展开时也是相当有用的——一个宏的展开代码里可以含有一个**FLET**或**LABELS**，用来创建可在宏主体中使用的函数。这个技术既可用来引入宏的用户要调用的函数，也可以只作为一种组织宏所生成的代码的方式。举个例子，这就是为什么能够定义出像**CALL-NEXT-METHOD**这种只能在一个方法的定义内使用的函数的原因。

一个与**FLET**和**LABELS**紧密相关的特殊操作符是**MACROLET**，它可以用来定义局部宏。局部宏的工作方式与**DEFMACRO**定义的全局宏一样，只是并不作用在全局名字空间上。求值一个**MACROLET**宏时，主体形式在局部宏定义生效的情况下被求值，其中的局部宏定义可能会覆盖全局的函数和宏定义，或是来自外围形式的局部定义。与**FLET**和**LABELS**一样，**MACROLET**可以被直接使用，也适用于宏生成的代码——通过将一些用户提供的代码包装进一个**MACROLET**，一个宏可以提供只用于这些代码中的构造，或是覆盖一个全局定义的宏。你将在第31章里看到**MACROLET**的后一种用法。

最后，还有一个定义宏的特殊操作符**SYMBOL-MACROLET**，它定义了一种名副其实的称为符号宏（symbol macro）的特殊类型的宏。符号宏和常规的宏相似，只是不能接受任何参数，并且只能用单个符号而非列表的形式来引用它。换句话说，当定义了一个特定名字的符号宏以后，在值的位置上对该符号的任何使用将被展开，由此产生的形式将在该位置上进行求值。这就是诸如

[1] 听起来可能令人惊讶，但确实有可能使匿名函数成为递归的。不过，你必须使用一种称为“Y组合器”(Y combinator)的古怪手法。Y组合器属于一种有趣的理论结果，并非实用的编程工具，因此完全在本书的讨论范围之外。

WITH-SLOTS和WITH-ACCESSORS是如何定义“变量”用来在特定范围内访问某个特定对象的状态的。例如，下面的WITH-SLOTS形式：

```
(with-slots (x y z) foo (list x y z)))
```

可以展开成使用SYMBOL-MACROLET的下列代码：

```
(let ((#:g149 foo))
  (symbol-macrolet
      ((x (slot-value #:g149 'x))
       (y (slot-value #:g149 'y))
       (z (slot-value #:g149 'z)))
    (list x y z)))
```

当表达式(list x y z)被求值时，符号x、y和z将被替换成它们的展开式，例如(slot-value #:g149 'x)。[①]

符号宏通常都是局部的，由SYMBOL-MACROLET定义，Common Lisp也提供了一个宏DEFINE-SYMBOL-MACRO来定义全局的符号宏。一个由SYMBOL-MACROLET定义的符号宏将覆盖由DEFINE-SYMBOL-MACRO或外围SYMBOL-MACROLET形式所定义的其他同名符号宏。

20.3　局部控制流

接下来将讨论的四个特殊操作符也会在词法环境中创建并使用名字，目的是为了调整控制流而非定义新的函数和宏。我曾经提到过这四个特殊操作符，因为它们提供了其他语言特性用到的底层机制。它们是BLOCK、RETURN-FROM、TAGBODY和GO。前两个操作符BLOCK和RETURN-FROM一起使用时，可以立即从一段代码中返回我在第5章里讨论过的将RETURN-FROM作为一种从函数中立即返回的方式，但它还有更一般的用途。另外的TAGBODY和GO提供了一种相当底层的goto结构，这种结构是目前你所见到的所有更上层循环结构的基础。

BLOCK形式的基本结构如下所示：

```
(block name
  form*)
```

其中的*name*是一个符号，而*form*是一些Lisp形式。这些形式按顺序进行求值，最后那个形式的值作为整个BLOCK的值返回，除非有一个RETURN-FROM用来从块结构中提前返回。如同你在第5章里看到的，一个RETURN-FROM形式由打算返回到该形式的块名称，以及一个可选的提供了返

① WITH-SLOTS不一定非要用SYMBOL-MACROLET来实现。在某些实现里，WITH-SLOTS可能会遍历提供给它的代码，并生成一个带有x、y和z的，已经被替换成对应的SLOT-VALUE形式的展开式。你可以通过求值下面的形式来查看你所用的实现是怎样做的：

```
(macroexpand-1 '(with-slots (x y z) obj (list x y z)))
```

不过，遍历形式体这件事由Lisp实现来做比用户代码更容易一些。要想让x、y和z仅在作为值出现时才被替换掉，这需要一个代码遍历器能够理解所有的特殊操作符，并且可以递归地展开所有宏形式来检测其展开式里是否含有值位置上的那些符号。Lisp实现本身显然带有它自己的一个代码遍历器，但这是Lisp中没有暴露给语言用户的少数部分之一。

回值的形式所组成。当一个RETURN-FROM被求值时，它会导致该命名的BLOCK立即返回。如果调用RETURN-FROM时带有返回值，那么BLOCK将返回该值；否则整个BLOCK将求值为NIL。

一个块的名字可以是任何符号，包括NIL在内。许多标准控制构造宏，诸如DO、DOTIMES和DOLIST，都会生成一个含有名为NIL的BLOCK的扩展。这允许你使用RETURN宏来从这些循环中跳出，该宏是(return-from nil ...)的语法糖。这样，下面的循环将打印出至多10个随机数，并在首次遇到大于50的数字时立即停下来：

```
(dotimes (i 10)
  (let ((answer (random 100)))
    (print answer)
    (if (> answer 50) (return))))
```

另一方面，诸如DEFUN、FLET和LABELS这类可以定义函数的宏，会将它们的函数体封装在一个与该函数同名的BLOCK中。这就是你可以用RETURN-FROM来从一个函数中返回的原因。

TAGBODY和GO之间的关系类似于BLOCK和RETURN-FROM的关系：一个TAGBODY形式定义了一个上下文，其中定义的名字可被GO使用。一个TAGBODY形式的模板如下所示：

```
(tagbody
  tag-or-compound-form*)
```

其中每个*tag-or-compound-form*要么是一个称为标记（tag）的符号，要么是一个非空的列表形式。这些列表形式按顺序进行求值，而那些标记将被忽略，除了我即将讨论的一种情况。当TAGBODY的最后一个形式被求值以后，整个TAGBODY返回NIL。在TAGBODY的词法作用域的任何位置，你可以使用GO特殊操作符立即跳转到任何标记上，而求值过程将从紧跟着该标记的那个形式开始继续进行。例如，你可以像下面这样使用TAGBODY和GO编写一个简单的无限循环：

```
(tagbody
 top
   (print 'hello)
   (go top))
```

注意，尽管标记名必须出现在TAGBODY的最顶层，而不能内嵌到其他形式中，但GO特殊操作符却可以出现在TAGBODY作用域的任何位置上。这意味着可以像下面这样编写一个随机次数的循环：

```
(tagbody
 top
   (print 'hello)
   (when (plusp (random 10)) (go top)))
```

还有一个更无聊的TAGBODY示例，它表明你可以在单个TAGBODY里使用多个标记，看起来像这样：

```
(tagbody
 a (print 'a) (if (zerop (random 2)) (go c))
 b (print 'b) (if (zerop (random 2)) (go a))
 c (print 'c) (if (zerop (random 2)) (go b)))
```

上面这个形式将不断作随机跳转并顺便打印出一些a、b和c，直到最后一个RANDOM表达式偶然返回了1并且控制落到了TAGBODY的结尾。

很少直接使用TAGBODY，因为使用已有的循环宏来编写迭代控制构造往往更方便。不过，它在将来自其他语言的算法转译成Common Lisp时会很有用，无论是自动的还是手工的。一个自动转译工具的例子是从FORTRAN到Common Lisp的转译器f2cl，它将FORTRAN源代码转译成Common Lisp，从而允许Common Lisp程序员得以使用各种各样的FORTRAN库。由于许多FORTRAN库写于结构化编程革命以前，所以代码里有很多跳转（goto）语句。f2cl编译器可以简单地将那些跳转语句转译成带有适当TAGBODY的GO语句。[①]

类似地，TAGBODY和GO在转译那些以文字或框图描述的算法时也很有用。例如，在Donald Knuth不朽的经典系列著作《计算机程序设计艺术》中，他使用了一种“菜谱”式的格式来描述算法：第一步，做这个；第二步，做那个；第三步，回到第二步；诸如此类。比如，在《计算机程序设计艺术，卷2：半数值算法》第3版（Addison-Wesley, 1998）的第142页，他以下面的形式描述了算法S，该算法将在第27章里用到：

算法 S （选择取样技术）：从有N个记录的集合里随机选出n个记录，其中$0 < n \leqslant N$。

S1. [初始化] 设$t \leftarrow 0$，$m \leftarrow 0$。（在本算法中，m表示已选出的记录数，t表示已经处理的输入记录的总数。）

S2. [生成U] 生成一个随机数U，它平均分布在0和1之间。

S3. [测试] 如果$(N-t)\,U \geqslant n-m$，那么转向步骤S5。

S4. [选择] 选择下一个记录，并将m和t递增1。如果$m < n$，那么转向步骤S2；否则取样过程结束，算法终止。

S5. [跳过] 跳过下一个记录（不将它选作样本），将t递增1，然后回到步骤S2。

这些描述可以轻易转译成一个Common Lisp函数，在重命名了一些变量以后，如下所示：

```
(defun algorithm-s (n max) ; max is N in Knuth's algorithm
  (let (seen                    ; t in Knuth's algorithm
        selected                ; m in Knuth's algorithm
        u                       ; U in Knuth's algorithm
        (records ()))           ; the list where we save the records selected
    (tagbody
     s1
       (setf seen 0)
       (setf selected 0)
     s2
       (setf u (random 1.0))
     s3
```

① 某个版本的f2cl现在是Common Lisp Open Code Collection (CLOCC) 的一部分，请查阅http://clocc. sourceforge. net/。相比之下，看看f2j（FORTRAN到Java的转译器）的作者被迫采取的方法吧。尽管Java虚拟机（JVM）支持一个跳转指令，但它没有直接暴露给Java。因此为了编译FORTRAN的跳转语句，他们首先将FORTRAN代码编译成带有那些对代表标签和跳转的空类的调用的合法Java源代码，然后对产生的字节码进行后期处理，把那些空调用再转译成JVM层面的字节码。这是很聪明的做法，但是太折磨人了。

```
    (when (>= (* (- max seen) u) (- n selected)) (go s5))
  s4
    (push seen records)
    (incf selected)
    (incf seen)
    (if (< selected n)
        (go s2)
        (return-from algorithm-s (nreverse records)))
  s5
    (incf seen)
    (go s2))))
```

这不算是精美的代码，但很容易验证它是Knuth算法的一个忠实转译。这些代码和Knuth的文字描述的不同之处在于，它是可以运行和测试的。然后你可以开始着手重构它，确保该函数在每次变更之后仍然可以工作。[①]

在经过一番优化以后，你可能最终会得到类似下面的代码：

```
(defun algorithm-s (n max)
  (loop for seen from 0
     when (< (* (- max seen) (random 1.0)) n)
     collect seen and do (decf n)
     until (zerop n)))
```

尽管一眼看不出来这些代码是否正确实现了算法S，但如果它是一系列与最初的Knuth算法描述的字面转译具有相同行为的函数，那么你有理由相信它是正确的。

20.4 从栈上回退

从语言的另一方面讲，特殊操作符还可以让你控制调用栈的行为。例如，尽管可以正常使用**BLOCK**和**TAGBODY**来管理单一函数内的控制流，但也可以将它们与闭包一起使用，可以强制从栈底部的函数立即非本地返回。这是因为**BLOCK**名字和**TAGBODY**标记可以被**BLOCK**或**TAGBODY**词法作用域之内的任何代码所闭合。例如，考虑下面这个函数：

```
(defun foo ()
  (format t "Entering foo~%")
  (block a
    (format t " Entering BLOCK~%")
    (bar #'(lambda () (return-from a)))
    (format t " Leaving BLOCK~%"))
  (format t "Leaving foo~%"))
```

传递给bar的匿名函数使用**RETURN-FROM**从**BLOCK**中返回。但**RETURN-FROM**要直到匿名函数被**FUNCALL**或**APPLY**调用时才会求值。现在假设bar函数看起来像这样：

---

① 由于这个算法取决于**RANDOM**所返回的值。你也许想要使用一致的随机数种子来测试它，这可以通过在每一次对algorithm-s的调用中将***RANDOM-STATE***绑定到(make-random-state nil)的值上来实现。例如，你可以通过求值下面的形式来对algorithm-s做一次基本的健全性检查：
(let ((*random-state* (make-random-state nil))) (algorithm-s 10 200))
如果你的重构都是合法的，那么这个表达式应当每次求值都得到同样的列表。

```
(defun bar (fn)
  (format t "  Entering bar~%")
  (baz fn)
  (format t "  Leaving bar~%"))
```

匿名函数仍然不会被调用。现在再看baz：

```
(defun baz (fn)
  (format t "   Entering baz~%")
  (funcall fn)
  (format t "   Leaving baz~%"))
```

最终，函数被调用了。但是一个位于栈的上方数层的**BLOCK**对于**RETURN-FROM**来说算什么呢？看起来一切工作正常——栈被回退到**BLOCK**最初建立的地方，且控制则从**BLOCK**返回。函数foo、bar和baz中的**FORMAT**表达式显示了这点：

```
CL-USER> (foo)
Entering foo
 Entering BLOCK
  Entering bar
   Entering baz
Leaving foo
NIL
```

注意，唯一打印出的“Leaving ...”信息是foo函数中出现在**BLOCK**之后的那个。

由于块的名字是词法作用域的，一个**RETURN-FROM**总是从它所在的词法环境中最小的外围**BLOCK**上返回，即使**RETURN-FROM**是在不同的动态上下文中执行的。例如，bar也可以含有一个名为a的**BLOCK**，像这样：

```
(defun bar (fn)
  (format t "  Entering bar~%")
  (block a (baz fn))
  (format t "  Leaving bar~%"))
```

这个额外的**BLOCK**根本不会改变foo的行为——名字a是词法解析的，并且是在编译期而非动态地解析的，因此这个插入的块对于**RETURN-FROM**的行为没有影响。反过来说，只有当**RETURN-FROM**出现在该**BLOCK**的词法作用域内部时，才可以使用一个**BLOCK**的名字。没有办法让块外的语句从该块上返回，除非通过调用一个在**BLOCK**的词法作用域内部封装的闭包。

TAGBODY和**GO**在这一点上与**BLOCK**和**RETURN-FROM**的工作方式相同。当调用一个含有**GO**形式的闭包时，如果这个**GO**被求值，那么栈将回退到适当的**TAGBODY**，然后跳转到特定的标记上。

不过，**BLOCK**名字和**TAGBODY**标记在某个重要方面与词法变量绑定不同。如同我在第6章讨论的，词法绑定具有无限时效（indefinite extent），这意味着即便在绑定形式返回后绑定也可以保持效果。另一方面，**BLOCK**和**TAGBODY**具有动态时效（dynamic extent）——只有当**BLOCK**和**TAGBODY**在栈上时，你才能通过**RETURN-FROM**回到一个**BLOCK**，或者通过**GO**回到一个**TAGBODY**标记上。换句话说，一个捕捉了块名或是**TAGBODY**标记的闭包只能向栈的下方传递从而留到稍后再调用，但它不能向栈的上方传递。如果你调用了一个闭包，它试图在某个**BLOCK**本身返回以后再通过**RETURN-FROM**回到这个**BLOCK**上，那么你将得到一个错误。同样，试图通过**GO**回到一个不存在

的TAGBODY上也将导致错误发生。[1]

你不太可能亲自使用BLOCK和TAGBODY来实现这种栈回退。但无论你何时使用状况系统，恐怕都是在间接地使用它们，因此理解其工作方式有助于你更好地理解它们，比如，当调用一个再启动时究竟发生了什么。[2]

CATCH和THROW是另一对可以强制回退栈的特殊操作符。相比到目前为止提到的其他相关操作符，这些操作符使用得更少，它们是早期没有Common Lisp状况系统的Lisp方言所留下的东西。绝对不能把它们跟诸如Java和Python这些语言中的try/catch和try/except结构相混淆。

CATCH和THROW是BLOCK和RETURN-FROM的动态版本。就是说，你用CATCH包装了一个代码体，然后用THROW使CATCH形式立即从一个特定值返回。区别在于，CATCH和THROW之间的关联是动态建立的——相对一个词法作用域的名字来说，一个CATCH的标签是对象，称为捕捉标记(catch tag)，而任何在CATCH的动态时效中求值的THROW在抛出该对象时，将导致栈回退到CATCH形式上，然后它会立即返回。这样，你可以编写另一个版本的foo、bar和baz函数，像下面这样使用CATCH和THROW来代替BLOCK和RETURN-FROM：

```
(defparameter *obj* (cons nil nil)) ; i.e. some arbitrary object

(defun foo ()
  (format t "Entering foo~%")
  (catch *obj*
    (format t " Entering CATCH~%")
    (bar)
    (format t " Leaving CATCH~%"))
  (format t "Leaving foo~%"))

(defun bar ()
  (format t "  Entering bar~%")
  (baz)
  (format t "  Leaving bar~%"))

(defun baz ()
  (format t "   Entering baz~%")
  (throw *obj* nil)
  (format t "   Leaving baz~%"))
```

注意，没有必要向下传递闭包，baz可以直接调用THROW。结果和之前的版本很相似。

```
CL-USER> (foo)
Entering foo
 Entering CATCH
  Entering bar
   Entering baz
Leaving foo
NIL
```

① 这是一个相当合理的限制——从一个已经返回了的形式中返回的意义并不是完全清楚的——当然，除非你是一个Scheme程序员。Scheme支持续延（continuation），一个允许从相同的函数调用中多次返回的语言构造。但出于多种原因，很少有Scheme之外的语言支持这类续延特性。

② 如果你是那种凡事都要刨根问底的人，那么思考一下怎样才能通过BLOCK、TAGBODY、闭包和动态变量来实现状况系统的那些宏，这可能是相当有意义的。

不过，CATCH和THROW过于动态了。在CATCH和THROW中，标记形式都会被求值，这意味着它们的值都是在运行期检测的。这样，如果在bar中的某些代码重新赋值或绑定了*obj*，那么baz中的THROW将不会抛出同样的CATCH。这使得代码中的CATCH和THROW比BLOCK和RETURN-FROM更难理解。使用了CATCH和THROW的foo、bar和baz的演示代码的唯一优势就是，不再需要向下传递一个闭包以便底层代码可以从一个CATCH中返回——任何在一个CATCH的动态时效内运行的代码都可以通过抛出正确的对象来返回。

在那些没有任何类似Common Lisp状况系统机制的古老Lisp方言里，CATCH和THROW用于错误处理。不过，为了确保它们的可管理性，捕捉标记通常只是一些引用了的符号，因此你只需观察代码就可以看出CATCH和THROW是否会在运行期关联在一起。在Common Lisp中，你很少有机会使用CATCH和THROW，因为使用状况系统会更加灵活。

最后一个跟栈控制有关的特殊操作符是我之前提到过的操作符UNWIND-PROTECT。UNWIND-PROTECT让你能够控制在栈被回退时所发生的事——确保特定代码在控制离开UNWIND-PROTECT作用域的任何情况下总可以运行，无论是通过一个被调用的再启动正常返回，还是采用任何在本章中所讨论的方式。[①]UNWIND-PROTECT的基本结构看起来像这样：

```
(unwind-protect protected-form
  cleanup-form*)
```

单一的*protected-form*被求值，随后无论它是否返回，*cleanup-form*都会被求值。如果*protected-form*正常返回了，那么它所返回的值将在执行这些用于清理的形式后被UNWIND-PROTECT返回。这些清理形式在与UNWIND-PROTECT相同的动态环境中被求值，因此，那些在进入UNWIND-PROTECT之前相同的动态变量绑定、再启动和状况处理器将对清理形式中的代码是可见的。

你偶尔会直接使用UNWIND-PROTECT。不过更常见的情况是将它作为WITH-风格宏的基础，类似于WITH-OPEN-FILE，它在上下文中求值任意数量的形式，其中它们所访问的某些资源需要在它们结束访问后被清理干净，无论它们是正常返回的、通过再启动返回的，还是其他的非本地退出。举个例子，如果你正在编写一个数据库，其中定义了函数open-connection和close-connection，你可能会写一个像下面这样的宏：[②]

```
(defmacro with-database-connection ((var &rest open-args) &body body)
  `(let ((,var (open-connection ,@open-args)))
     (unwind-protect (progn ,@body)
       (close-connection ,var))))
```

它可以让你写出类似下面这样的代码：

```
(with-database-connection (conn :host "foo" :user "scott" :password "tiger")
  (do-stuff conn)
  (do-more-stuff conn))
```

① UNWIND-PROTECT本质上与Java和Python中的try/finally结构等价。

② 事实上，CLSQL是跨Lisp平台和数据库的SQL接口库，它确实提供了一个称为with-database的类似的宏。CLSQL的主页是http://clsql.b9.com。

你不必担心数据库的关闭，因为UNWIND-PROTECT会确保数据库被关闭，不论with-database-connection形式的主体中发生了什么。

20.5 多值

Common Lisp的另一个特性是我在第11章里讨论GETHASH时提到过的，即单一形式可以返回多个值的能力。现在我将进一步讨论它的细节。不过，把这些内容放在关于特殊操作符的章节里并不太合适，因为多重返回值并不仅仅是由一两个特殊操作符提供的，而是紧密集成到了整个语言之中。在处理多值时，最常使用的操作符是宏和函数，而非特殊操作符，但最后获取多重返回值的基本功能是由特殊操作符MULTIPLE-VALUE-CALL提供的，而更常用的MULTIPLE-VALUE-BIND宏则构建于其上。

理解多重返回值的关键在于，返回多个值与返回一个列表是有本质不同的——如果一个形式返回了多个值，除非你做了一些特别的事情来捕捉多个值，那么除了主值（primary value）以外的其他值都将被悄悄地丢掉。为了理解这一区别，来看看函数GETHASH，它返回两个值：哈希表中找到的值和一个布尔值——在没有找到值时为NIL。如果它将这两个值返回到一个列表中，那么每次你调用GETHASH时都必须解析这个列表来取得实际的值，无论你是否关心第二个返回值。假设你有一个哈希表*h*，它含有一些数值。如果GETHASH返回一个列表，那么你就不能写出类似下面的形式了：

```
(+ (gethash 'a *h*) (gethash 'b *h*))
```

因为“+”期待其参数是数字而非列表。但由于多重返回值机制悄悄地丢弃了那个不需要的第二个返回值，从而使这个形式可以正常工作。

使用多重返回值包括两个方面——返回多个值以及获取那些返回多值的形式所返回的非主值。返回多值的开始点是函数VALUES和VALUES-LIST。这些都是正规函数而非特殊操作符，因此它们的参数将以正常方式传递。VALUES接受可变数量的参数并将它们作为多值返回，VALUES-LIST接受单个列表并将它的元素作为多值返回。换句话说：

```
(values-list x) ≡ (apply #'values x)
```

多值返回的机制和向函数传递参数的机制一样，都是具体实现相关的。几乎所有可以返回一些子形式的值的语言构造都会“传递”多值，并返回由其子形式返回的所有值。这样，一个返回了调用VALUES或VALUES-LIST的结果的函数将返回多值，并且其结果来自对这个函数的调用的另一个函数也会返回多值，以此类推。[①]

但是当一个形式被放在值的位置上求值时，只有主值会被使用，这也就是之前的加法形式能

① 少量有用的宏并不会传递它们所求值形式的其他返回值。特别的是PROG1宏，它像PROGN那样求值一组形式并返回第一个形式的值，只返回该形式的主值。同样，PROG2返回其第二个子形式的值，也只返回主值。特殊操作符MULTIPLE-VALUE-PROG1是PROG1的一个变体，它可以返回第一个形式的所有值。PROG1不具有MULTIPLE-VALUE-PROG1那样的行为，这多少算是一个缺点，但这两个宏其实都不太常用。OR和COND宏也并不总是对多值透明的，在特定的子形式上只返回其主值。

够以你期待的方式运行的原因。特殊操作符**MULTIPLE-VALUE-CALL**提供了访问一个形式返回的多个值的机制。**MULTIPLE-VALUE-CALL**和**FUNCALL**相似，除了**FUNCALL**是个正规函数并且因此只能看到并传递那些传给它的主值，而**MULTIPLE-VALUE-CALL**则为它第一个子形式返回的函数传递其余子形式返回的所有的值。

```
(funcall #'+ (values 1 2) (values 3 4))              → 4
(multiple-value-call #'+ (values 1 2) (values 3 4)) → 10
```

不过，你一般不需要简单地将一个函数返回的所有值都传给另一个函数。更常见的用法是，你希望多个值分别保存在不同的变量里，然后再对这些变量作处理。第11章里提到的**MULTIPLE-VALUE-BIND**宏就是最常用的用于接收多重返回值的操作符。它的模板看起来像这样：

```
(multiple-value-bind (variable*) values-form
  body-form*)
```

其中*values-form*被求值，它返回的多个值被绑定到那些变量上。然后那些*body-form*在绑定的作用下被求值。这样：

```
(multiple-value-bind (x y) (values 1 2)
  (+ x y)) → 3
```

另一个宏**MULTIPLE-VALUE-LIST**甚至更简单——它接受单一的形式，求值它，然后将得到的多个值收集到一个列表中。换句话说，它是**VALUES-LIST**的逆操作。

```
CL-USER> (multiple-value-list (values 1 2))
(1 2)
CL-USER> (values-list (multiple-value-list (values 1 2)))
1
2
```

不过，如果你发现自己使用了很多的**MULTIPLE-VALUE-LIST**，这也许是一个信号，表明某些函数应当开始返回一个列表而不是多值了。

最后，如果你想要将一个形式返回的多个值一次性赋值到已有变量上，那么可以将**VALUES**作为可**SETF**的位置来使用。例如：

```
CL-USER> (defparameter *x* nil)
*X*
CL-USER> (defparameter *y* nil)
*Y*
CL-USER> (setf (values *x* *y*) (floor (/ 57 34)))
1
23/34
CL-USER> *x*
1
CL-USER> *y*
23/34
```

20.6 EVAL-WHEN

为了写出某些特定类型的宏，你必须理解**EVAL-WHEN**操作符。出于一些原因，Lisp书籍通常将**EVAL-WHEN**视为巫师级别的话题。但其实理解**EVAL-WHEN**的唯一前提，只是理解两个函数**LOAD**

和COMPILE-FILE是如何交互的。理解EVAL-WHEN对于你开始编写特定类型的更加专业的宏非常重要，例如你将在第24章和第31章里编写的一些宏。

我在前面的章节已经简要提过LOAD和COMPILE-FILE之间的关系，这里有必要再说一次。LOAD的任务是加载一个文件并求值它包括的所有顶层形式。COMPILE-FILE的任务则是将一个源代码文件编译成FASL文件，后者随后可以被LOAD加载，因此(load "foo.lisp")和(load "foo.fasl")在本质上是等价的。

由于LOAD在读取每一个形式以后立即求值，因而求值文件中靠前面的形式就会影响读取和求值后续形式的行为。例如，求值一个IN-PACKAGE形式将改变*PACKAGE*的值，这将影响读取后续形式的方式。[1]类似地，一个较早出现在文件中的DEFMACRO形式可以定义一个可被文件中后续代码使用的宏。[2]

另一方面，COMPILE-FILE通常在编译时不求值文件中的形式。只有当FASL文件被加载时，这些形式或它们的编译后等价物才会被求值。尽管如此，COMPILE-FILE必须求值一些诸如IN-PACKAGE和DEFMACRO等形式，以保持(load "foo.lisp")和(load "foo.fasl")具有一致的行为。

那么诸如IN-PACKAGE和DEFMACRO这样的宏在被COMPILE-FILE处理时是怎样工作的呢？在Common Lisp标准之前的Lisp版本里，文件编译器简单地在编译后进一步求值某些形式。Common Lisp从MacLisp中借鉴了EVAL-WHEN，从而避免了类似异常情况的发生。顾名思义，这个操作符允许你控制特定的代码在何时被求值。EVAL-WHEN形式的模板看起来像这样：

```
(eval-when (situation*)
  body-form*)
```

存在三种可能的情形：:compile-toplevel、:load-toplevel和:execute，并且你指定的那些情形将控制所有*body-form*的求值时间。一个带有多重情形的EVAL-WHEN等价于分开的几个EVAL-WHEN形式，同样的代码每种情形各一个。为了解释三种情形的含义，我需要解释一下COMPILE-FILE，它也称为文件编译器，用来编译一个文件。

为了解释COMPILE-FILE是如何编译EVAL-WHEN形式的，先要介绍编译顶层形式和非顶层形式的区别。简单地说，一个顶层形式就是一些编译之后可以在FASL文件加载时运行的代码。这样，所有直接出现在一个源代码文件顶层的形式都将作为顶层形式来编译。类似地，任何直接出现在一个顶层PROGN中的形式也将作为顶层形式来编译，因为PROGN本身并不做任何事——它只

---

① 加载一个带有IN-PACKAGE形式的文件，在LOAD返回以后看不到*PACKAGE*值的改变，这是因为LOAD在对该变量做任何改变之前绑定了它的当前值。换句话说，一些等价于下面这个LET形式的结构封装了LOAD中其余的代码：

```
(let ((*package* *package*)) ...)
```

任何对*PACKAGE*的赋值都将是新的绑定，而旧的绑定将在LOAD返回时恢复。它还以同样方式绑定了变量*READTABLE*，该变量我尚未谈及。

② 在某些实现中，你或许可以正确求值一个在函数体中使用了未定义宏的DEFUN定义，只要在函数实际被调用之前定义好该宏即可。但仅当从源代码加载这些定义时，它们才可以正常工作，而在通过COMPILE-FILE编译时是不行的。因此一般来说，宏定义必须在它们被使用前被求值。

是把其子形式组织在一起，然后它们会在FASL被加载时运行。[1]类似地，直接出现在一个MACROLET或SYMBOL-MACROLET中的形式将作为顶层形式来编译，因为在编译器展开了这些局部宏或符号宏之后，编译后的代码中就不再有MACROLET或SYMBOL-MACROLET了。最后，一个顶层宏形式的展开式将作为顶层形式来编译。

这样，一个出现在源代码文件顶层的DEFUN就是一个顶层形式，那些定义了函数并且将其与函数名相关联的代码将会在加载FASL时运行——但是函数体中的形式不是顶层形式，它们要等到函数被调用时才会运行。大多数形式在作为顶层和非顶层形式来编译时产生的结果都是相同的，但一个EVAL-WHEN的语义取决于它是作为顶层形式还是非顶层形式来编译的，或是简单地被求值，所有这些条件将按照列在其情形列表中的情形组合来决定。

当一个EVAL-WHEN作为顶层形式来编译时，情形:compile-toplevel和:load-toplevel控制其含义。当存在:compile-toplevel时，文件编译器将在编译期求值其子形式。当存在:load-toplevel时，它将把那些子形式作为顶层形式来编译。如果这两个情形都不在顶层EVAL-WHEN的情形列表中，那么编译器将会忽略它。

当一个EVAL-WHEN作为非顶层形式来编译时，它要么在:execute情形被指定时像PROGN那样被编译，要么被忽略。类似地，一个被求值的EVAL-WHEN（包括那些被LOAD加载的源代码文件中的顶层EVAL-WHEN，以及出现在带有:compile-toplevel情形的顶层EVAL-WHEN的子形式中的在编译期被求值的EVAL-WHEN），要么在:execute存在时被当作PROGN来对待，要么被忽略。

这样，一个诸如IN-PACKAGE这样的宏可以通过展开成类似下面这样的形式，来确保同时在编译期和源代码加载期都产生效果：

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf *package* (find-package "PACKAGE-NAME")))
```

*PACKAGE*在编译期设置是因为有:compile-toplevel情形，在加载FASL时设置是因为有:load-toplevel，而在加载源代码时设置是因为有:execute。

你会经常在两种情况下使用EVAL-WHEN。一种是你想编写一个需要在编译期保存一些信息的宏时，这些信息将被同一个文件中的其他宏形式的展开式所用到。通常这些都是定义性的宏：一个文件开始处的定义可以影响同一个文件内其他定义所生成的代码。你将在第24章里编写这种类型的宏。

另一个你可能需要EVAL-WHEN的场合是，你想要把一个宏和它的辅助函数与使用该宏的代码放在同一个文件里。DEFMACRO已经在其展开式里包含了一个EVAL-WHEN，因此宏定义可以立即被文件的后续部分所使用。但是DEFUN正常情况下并不会在编译期产生函数定义。但如果你在定义了一个宏的文件中使用这个宏，就需要确保被用到的宏和该宏所用到的函数都有定义。如果你把该宏所用到的任何函数的DEFUN都封装在一个带有:compile-toplevel的EVAL-WHEN里，那么在宏的展开函数运行时这些定义就可以使用了。你很可能想再加上:load-toplevel和:execute，因为

① 相反，一个顶层LET中的子形式并不会作为顶层形式来编译，因为它们不会在FASL加载时直接运行。它们将会运行，但是在由LET所建立的绑定和运行期上下文中运行的。理论上来说，一个不绑定任何变量的LET将会被当作PROGN来对待，但其实不是这样的，出现在LET中的子形式不会作为顶层形式来对待。

在文件被编译加载或者从源代码不编译而直接加载以后，该宏也需要这些函数。

20.7 其他特殊操作符

其余的4个特殊操作符分别是**LOCALLY**、**THE**、**LOAD-TIME-VALUE**和**PROGV**，它们都允许你访问以其他任何方式都无法访问到的语言的底层部分。**LOCALLY**和**THE**是Common Lisp声明系统的一部分，它们用来与编译器沟通而对代码的含义没有影响，但可能有助于编译器生成更好的代码，比如更快更清晰的错误信息。[1]我将在第32章里简要地讨论有关声明的内容。

另外两个**LOAD-TIME-VALUE**和**PROGV**都很少会用到，而且解释为什么会想要使用它们，比解释它们能干什么还费劲。因此我只告诉你它们能干什么，让你知道它们的存在。日后当你偶尔遇到刚好可以用上它们的场合时，就知道该怎么用了。

顾名思义，**LOAD-TIME-VALUE**用来创建一个在加载期决定的值。当文件编译器编译含有**LOAD-TIME-VALUE**形式的代码时，它会安排在加载FASL时只求值其第一个子形式一次，然后让含有**LOAD-TIME-VALUE**形式的代码指向该值。换句话说，下面的写法：

```
(defvar *loaded-at* (get-universal-time))

(defun when-loaded () *loaded-at*)
```

可以简化成这样：

```
(defun when-loaded () (load-time-value (get-universal-time)))
```

在没有被**COMPILE-FILE**处理过的代码中，**LOAD-TIME-VALUE**仅在代码被编译时求值一次，这可能是在显式使用**COMPILE**编译一个函数，或是在求值代码的过程中具体实现进行了隐式的编译时发生的。在未编译的代码中，**LOAD-TIME-VALUE**在其每次被求值时都会求值其子形式。

最后，**PROGV**可以为变量创建其名字在运行期才确定的新的动态绑定。这对于支持动态作用域变量的语言实现嵌入式解释器尤其有用。其基本框架如下所示：

```
(progv symbols-list values-list
  body-form*)
```

其中*symbols-list*是一个形式，它求值到一个符号的列表上，而*values-list*是一个求值到值列表的形式。每个符号被动态地绑定到对应的值上，然后求值那些*body-form*。**PROGV**和**LET**的区别在于，*symbols-list*是在运行期求值的，被绑定的变量名可以动态地确定。要我说，这并不是你需要经常干的事情。

这就是特殊操作符的所有内容。在下一章里，我将回到更加实用的话题，展示如何使用Common Lisp的包系统来控制名字空间，这样你才可以写出彼此并存而没有名字冲突的库和应用程序。

① 唯一一个对程序语义有影响的声明是第6章里提到的**SPECIAL**声明。