第22章 高阶LOOP

第7章曾简要讨论了扩展形式的LOOP宏。正如当时提到的，LOOP本质上提供了用来编写迭代构造的专用语言。

这看起来可能有些无聊——发明一种全新的语言只为编写循环。但如果你深入思考循环结构在程序中的各种用法，就会发现事实上这样做是合理的。任何规模的程序都会包含相当数量的循环。虽然它们不都是完全一样的，但也不是完全不同的。循环里有很多固定的编码模式，尤其是在循环前后包含代码时——为循环做准备的模式，确保循环过程所需的模式，以及当循环结束时所需操作的模式。LOOP语言捕捉了这些模式，让你可以直接表达各种循环。

LOOP宏由许多部分组成——LOOP反对者们的一个主要论点就是说它过于复杂。在本章里，我将从头介绍LOOP，系统化地介绍它的多个组成部分以及彼此间配合使用的方式。

22.1 LOOP 的组成部分

你可以在一个LOOP中做下列事情：

- 以数值或多种数据结构为步长来做循环；
- 在循环的过程中收集、计数、求和以及求最大值或最小值；
- 执行任意Lisp表达式；
- 决定何时终止循环；
- 条件执行上述内容。

另外，LOOP还提供了用于下列事务的语法：

- 创建用于循环内部的局部变量；
- 指定任意Lisp表达式在循环开始前和结束后运行。

LOOP的基本结构是一个子句集合，其中每个子句都以一个循环关键字（loop keyword）开始。[1]每个子句被LOOP宏解析的方式取决于具体的关键字。第7章已经介绍了一些主要的关键字，包括for、collecting、summing、counting、do以及finally。

① “循环关键字”这个名字取得并不是很好，因为循环关键字并不是正常意义下的KEYWORD包中的关键字。事实上，来自任何包的任何符号，只要有适当的名字就可以了，LOOP宏只关心它们的名字。不过，通常情况下它们都被写成不带有包限定符的形式并在当前包下被读取（必要时会创建新符号）。

22.2 迭代控制

大多数所谓的迭代控制子句都以循环关键字for或是它的同义词as[1]开始，后接一个变量的名字。变量名后面的内容取决于for子句的类型。

一个for子句的下级子句可以对下列内容进行迭代：

- 数字范围（以指定的间隔向上或向下）；
- 由单独的项组成的列表；
- 构成列表的点对单元；
- 向量的元素（包括诸如字符串和位向量这样的向量子类型）；
- 哈希表的键值对；
- 一个包中的符号；
- 对给定形式反复求值得到的结果。

循环可以含有多个for子句，其中每个子句都可以命名其自己的循环变量。如果循环含有多个for子句，其中任何一个子句达到结束条件循环都会终止。例如，下面的循环：

```
(loop
  for item in list
  for i from 1 to 10
  do (something))
```

将迭代至多10次，但在list含有少于10项时会提前终止。

22.3 计数型循环

算术迭代子句可以控制循环体的执行次数，它通过在一个整数范围上步进来做到这点，每前进一步就执行一次循环体。这些子句由for（或as）之后紧跟下列介词短语中的1~3个构成：起始短语、终止短语以及步长短语。

起始短语指定了该子句的变量初始值。它由介词from、downfrom或upfrom之一后接一个提供初值（一个数字）的形式所构成。

终止短语指定了循环的终止点。它由介词to、upto、below、downto或above之一后接一个提供终值的形式所构成。当使用upto和downto时，循环体将在变量通过终止点时终止（通过以后不会再次求值循环体）；而当使用below和above时，它会提前一次迭代终止循环。

步长短语由介词by和一个形式所构成，该形式必须求值为一个正数。变量将按照该数在每次迭代时步进（向上或向下，取决于其他短语）或是在其默认时每次步进1。

你必须至少指定一个上述介词短语。默认值是从零开始，每次迭代时加1，然后一直加下去，或者更有可能的是直到其他某个子句终止了循环。你可以通过添加适当的介词短语来修改这些默认值中的任何一个或全部。唯一需要注意的是如果要逐步递减的话，不存在默认的初始值，必须

① 因为当初LOOP的目标之一就是允许循环表达式可以被写成类似英语的语法，所以许多关键字都有一些同义词，它们对于LOOP来说处理方法相同，但在不同的语境下可以更接近英语的语法习惯。

使用from或downfrom来指定一个。因此，形式：

```
(loop for i upto 10 collect i)
```

会收集到11个整数（从零到十），而下面这个形式的行为则是未定义的：

```
(loop for i downto -10 collect i)          ; wrong
```

以下是一种正确的写法：

```
(loop for i from 0 downto -10 collect i)
```

另外要注意，由于**LOOP**是一个宏，运行在编译期，它需要能够完全基于这些介词而不是一些形式的值来决定变量步进的方向，因为形式的值到运行期才会知道。因此，形式

```
(loop for i from 10 to 20 ...)
```

可以工作得很好，因为默认就是递增步进。但形式：

```
(loop for i from 20 to 10 ...)
```

将不知道是从20向下数到10。更糟糕的是，它不会给你报错——由于i已经大于10了，所以循环根本不会执行。此时，你必须写成：

```
(loop for i from 20 downto 10 ...)
```

或是

```
(loop for i downfrom 20 to 10 ...)
```

最后，如果你只是想让一个循环重复特定的次数，那么可以将下列形式中的一个子句

```
for i from 1 to number-form
```

替换成如下所示的一个repeat子句：

```
repeat number-form
```

这些子句在效果上是等价的，只是repeat子句没有创建显式的循环变量。

22.4　循环集合和包

用于迭代列表的for子句比算术子句更简单一些。这种子句只支持两个介词短语，in和on。一个下列形式的短语

```
for var in list-form
```

将在求值*list-form*所产生的列表的所有元素上推动变量*var*。

```
(loop for i in (list 10 20 30 40) collect i)          → (10 20 30 40)
```

有时这个子句会在一个by短语的辅助下使用，by短语指定了一个用来在列表中向下移动的函数，其默认值是**CDR**，但它可以是任何接受一个列表并返回其子列表的函数。例如，可以像下面这样收集一个列表中相隔的元素：

```
(loop for i in (list 10 20 30 40) by #'cddr collect i) → (10 30)
```

on介词短语被用来在构成列表的点对单元上步进变量*var*。

```
(loop for x on (list 10 20 30) collect x) → ((10 20 30) (20 30) (30))
```

该短语也接受一个介词by：

```
(loop for x on (list 10 20 30 40) by #'cddr collect x) → ((10 20 30 40) (30 40))
```

循环向量（包括字符串和位向量）的元素与循环列表的元素类似，只是要使用介词across来代替in。[1]例如：

```
(loop for x across "abcd" collect x) → (#\a #\b #\c #\d)
```

迭代哈希表或包稍微复杂一些，因为哈希表和包可能包含需要迭代的不同值的集合——哈希表中的键或值，以及包中不同类型的符号。两种迭代都遵循相同的模式。基本的模式如下所示：

```
(loop for var being the things in hash-or-package ...)
```

对于哈希表来说，*things*的可能值是hash-keys和hash-values，它们使*var*与哈希表中连续的键或值绑定。*hash-or-package*形式只被求值一次并产生一个值，它必须是一个哈希表。

要想在包上迭代，*things*可以是symbols、present-symbols和external-symbols，它们使*var*被绑定到包的每个可访问的符号、当前存在的符号（即在包里创建的或导入进该包的符号），或是每一个从该包中导出的符号上。*hash-or-packge*被求值并产生一个包的名字（这会导致用**FIND-PACKAGE**来查找）或包对象本身。for子句的许多部分也可用同义词。在the的位置上可用each，你还可以用of来代替in，另外你也可以将*things*写成单数形式（例如，hash-key或symbol）。

最后，由于你经常需要在一个哈希表上同时迭代键和值，哈希表子句还在其结尾处支持using子句。

```
(loop for k being the hash-keys in h using (hash-value v) ...)
(loop for v being the hash-values in h using (hash-key k) ...)
```

这两个循环都可以将k绑定到哈希表的每个键上，再把v绑定到对应的值上。注意using子句的第一个元素必须写成单数形式。[2]

22.5 等价-然后迭代

如果其他的for子句都无法确切支持你所需要的变量步进形式，那么你可以通过等价-然后（equals-then）子句来完全控制步进的方式。这个子句跟**DO**循环中的绑定语句很相似，但更接近Algol语言的语法。完整的形式如下所示：

---

① 你可能想知道为什么**LOOP**不使用同样的介词，然后自己检查当前究竟是循环列表还是向量。这是**LOOP**作为一个宏所带来的另一个后果：列表或者向量的值直到运行期才会知道，但**LOOP**作为一个宏必须在编译期生成代码，并且**LOOP**的设计者们想要它生成极其高效的代码。为了能够生成用来访问的高效的代码，比如说一个向量，它需要在编译期知道这个值在运行期将是一个向量。因此，需要采用不同的介词。

② 不要问我为什么**LOOP**的设计者们没有使用不带括号的风格来表示using子句。

```
(loop for var = initial-value-form [ then step-form ] ...)
```

和通常一样，*var*是需要步进的变量名。它的初值在首次迭代之前通过求值*initial-value-form*而获取到。在每一次后续迭代中，*step-form*被求值，然后它的值成为了*var*的新值。如果子句中没有then部分，那么*initial-value-form*将在每次迭代中重新求值以提供新值。注意，这和一个没有步长形式的DO绑定子句的行为是不同的。

*step-form*可以引用其他的循环变量，包括由循环中其他后续for子句所创建的循环变量。例如：

```
(loop repeat 5
      for x = 0 then y
      for y = 1 then (+ x y)
      collect y) → (1 2 4 8 16)
```

不过，每个for子句都是以它们各自出现的顺序来逐个求值的，因此在前面的循环中，第二次迭代时x会在y改变（变成1）之前被设置为y的值。但是y随后会被设置为它的旧值（1）与x的新值之和。如果for子句的顺序反过来，那么结果将会改变。

```
(loop repeat 5
      for y = 1 then (+ x y)
      for x = 0 then y
      collect y) → (1 1 2 4 8)
```

不过，通常你都想让多个变量的步长形式在任何变量被赋予新值之前计算完毕（类似于DO步进其变量的方式）。在这种情况下，你可以将多个for子句连在一起，将除第一个以外的for全部替换成and。你已经在第7章通过LOOP计算Fibonacci的示例里见过它的公式了。这是基于前面两个例子的另一个变体：

```
(loop repeat 5
      for x = 0 then y
      and y = 1 then (+ x y)
      collect y) → (1 1 2 3 5)
```

22.6　局部变量

尽管循环所需要的主要变量通常都会在for子句中隐式声明，但有时也需要额外的变量，这些变量可以使用with子句来声明。

```
with var [ = value-form ]
```

*var*将成为一个局部变量的名字，它会在循环结束时被删除。如果with子句含有一个*= value-form*部分，那么变量将会在循环的首次迭代之前初始化为*value-form*的值。

多个with子句可以同时出现在一个循环里。每个子句将根据其出现的顺序独立求值，并且其值将在处理下一个子句之前完成赋值，从而允许后面的变量可以依赖于已经声明过的变量。完全无关的变量可以用and连接每个声明并放在一个with子句中。

22.7 解构变量

一个尚未谈及的LOOP宏的非常有用的特性，是其解构赋值给循环变量的列表值的能力。这可以让你取出赋值给循环变量的列表里的值，类似于DESTRUCTURING-BIND的工作方式但没有那么复杂。基本上，你可以将任何出现在for或with子句中的循环变量替换成一个符号树，这样，原本赋值到简单变量上的列表值将改为解构到以树中的符号所命名的变量上。一个简单的例子如下所示：

```
CL-USER> (loop for (a b) in '((1 2) (3 4) (5 6))
           do (format t "a: ~a; b: ~a~%" a b))
a: 1; b: 2
a: 3; b: 4
a: 5; b: 6
NIL
```

这棵树还可以包含带点的列表，这时点之后的名字将像一个&rest参数那样处理，被绑定到列表的其余元素上。这对于for/on类循环来说特别有用，因为值总是一个列表。例如，下面这个循环（第8章曾用它来输出一个逗号分隔的列表）：

```
(loop for cons on list
    do (format t "~a" (car cons))
    when (cdr cons) do (format t ", "))
```

也可以写成这样：

```
(loop for (item . rest) on list
    do (format t "~a" item)
    when rest do (format t ", "))
```

如果你想要忽略一个解构列表中的值，可以用NIL代替相应变量的名字。

```
(loop for (a nil) in '((1 2) (3 4) (5 6)) collect a) → (1 3 5)
```

如果解构列表含有比列表中的值更多的变量，那么多余的变量将被设置为NIL，这使得所有变量本质上都像是&optional参数。不过，没有任何跟&key参数等价的东西。

22.8 值汇聚

值汇聚（value accumulation）语句可能是LOOP中最有用的部分。尽管迭代控制语句提供了一个表示循环基本结构的简洁的语法，但它们本质上与DO、DOLIST和DOTIMES所提供的机制是等价的。

另一方面，值汇聚子句为循环过程中涉及值汇聚的常见循环用法提供了一套简洁表示法。每个汇聚子句都以一个动词后接下列模式开始：

```
verb form [ into var ]
```

每次通过循环时，汇聚子句会对*form*求值并将其按照由*verb*所决定的方式保存起来。通过into子句，这些值被保存在名为*var*的变量里。该变量对于循环来说是局部的，就像它是被一个

with子句所声明的那样。如果没有into下级子句，那么汇聚子句将汇聚出一个作为整个循环表达式返回值的默认值。

可用的动词包括collect、append、nconc、count、sum、maximize和minimize。还有它们对应的进行时形式的同义词：collecting、appending、nconcing、counting、summing、maximizing和minimizing。

collect子句会构造一个列表，列表中包含以代码中的顺序排列的所有*form*的值。这是一个特别有用的构造，因为手工编写一个像**LOOP**那样有效率的列表来收集代码非常困难。[①]与collect相关的动词是append和nconc。这两个动词都将值汇聚到一个列表上，但它们所汇聚的值本身也必须是列表，然后像函数**APPEND**或**NCONC**[②]那样将所有列表汇聚成单个列表。

其余的汇聚子句都用来汇聚数值。动词count统计*form*为真的次数，sum收集所有*form*的值之和，maximize收集它所看到的*form*的最大值，而minimize则收集最小值。例如，假设你定义了一个变量*random*，它含有一个随机数列表。

```
(defparameter *random* (loop repeat 100 collect (random 10000)))
```

那么下面的循环将返回关于这些数的一个含有多种统计信息的列表：

```
(loop for i in *random*
   counting (evenp i) into evens
   counting (oddp i) into odds
   summing i into total
   maximizing i into max
   minimizing i into min
   finally (return (list min max total evens odds)))
```

---

① 难点在于必须跟踪列表的尾部并通过**SETF**尾部的**CDR**来向列表添加新的点对。一个由(loop for i upto 10 collect i)所生成代码的手写等价版本如下所示：

```
(do((list nil) (tail nil) (i 0 (1+ i)))
   ((> i 10) list)
 (let ((new (cons i nil)))
   (if (null list)
       (setf list new)
       (setf (cdr tail) new))
   (setf tail new)))
```

当然，你很少需要编写像这样的代码。可以使用**LOOP**，也可以（如果出于某种原因你不想使用**LOOP**的话）使用标准的收集值的**PUSH/NREVERSE**。

② 回顾一下，**NCONC**是**APPEND**的破坏性版本——安全使用nconc子句的场合仅限于你正在收集的值都是全新的列表，而且该列表没有跟其他列表共享任何结构。例如，下面的代码是安全的：

```
(loop for i upto 3 nconc (list i i)) → (0 0 1 1 2 2 3 3)
```

而这个将给你带来麻烦：

```
(loop for i on (list 1 2 3) nconc i) → undefined
```

它将很可能进入一个无限循环，因为由(list 1 2 3)所产生的列表被破坏性地修改以指向它自身。其实这个行为也难以保证，因为其行为根本没有定义。

22.9 无条件执行

虽然值汇聚构造非常有用，但如果没有机会在循环体中执行任意代码的话，**LOOP**就不是一个很好的通用迭代机制了。

在一个循环之内执行任意代码的最简单方式是使用do子句。与之前讨论过的那些带有介词和进一步子句的其他子句相比，do具有一种Yoda式的简洁性。[①]一个do子句由单词do（或doing）后接一个或多个Lisp形式构成，这些形式将在do子句开始运行时全部被求值。do子句结束于一个循环的闭合括号或是下一个循环关键字。

例如，为了打印出数字1到10，可以这样来写：

```
(loop for i from 1 to 10 do (print i))
```

另一个更有趣的立即执行的形式是return子句。这个子句由单词return后接单个Lisp形式组成，当该形式被求值时，得到的结果将立即作为整个循环的值返回。

使用Lisp的常规控制流操作符，例如**RETURN**和**RETURN-FROM**，也可以从循环中的do子句里跳出。注意return子句总是从临近的**LOOP**表达式里返回，而do子句中的**RETURN**或**RETURN-FROM**可以从任意封闭的表达式中返回。举个例子，比较下列表达式

```
(block outer
  (loop for i from 0 return 100) ; 100 returned from LOOP
  (print "This will print")
  200) → 200
```

和

```
(block outer
  (loop for i from 0 do (return-from outer 100)) ; 100 returned from BLOCK
  (print "This won't print")
  200) → 100
```

上述do和return子句统称为无条件执行子句。

22.10 条件执行

由于do子句可以包含任意Lisp形式，所以可以在这里使用任何Lisp表达式，包括**IF**和**WHEN**这样的控制构造。下面就是只打印1到10之间所有偶数的一种循环的写法：

```
(loop for i from 1 to 10 do (when (evenp i) (print i)))
```

不过，有时你会需要循环子句层面上的条件控制。例如，假设你只想用一个summing子句来求和从1到10之间的偶数。你不可能用一个do子句来写出这样的循环，因为没有办法在一个正规的Lisp形式里“调用”sum i。对于类似这种情况，就需要用到LOOP自己的条件表达式，如下所示：

① “No!Try not. Do ... or do not. There is no try.”（不!不要试。要么做要么不做。没有机会可试）——Yoda，*The Empire Strikes Back*（《星球大战5》）。

```
(loop for i from 1 to 10 when (evenp i) sum i) → 30
```

LOOP提供了三种条件构造，它们全部遵循下面的基本模式：

```
conditional test-form loop-clause
```

其中的*conditional*可以是if、when或unless，*test-form*可以是任何正规Lisp形式，而*loop-clause*则可以是一个值汇聚子句（count、collect，等等）、一个无条件执行子句或是另一个条件执行子句。多个循环子句可以通过and连接成单一条件。

还有一点儿语法糖：在第一个循环子句里，测试形式之后，可以使用变量it来指代由测试形式所返回的值。例如，下面的循环可以收集在列表some-list中查找键时所找到的在哈希表some-hash中对应的非空值：

```
(loop for key in some-list when (gethash key some-hash) collect it)
```

条件子句在每次通过循环时都会执行。if或when子句会在它的*test-form*求值为真时执行其*loop-clause*。unless子句可以把测试反过来，仅当*test-form*为NIL时才执行*loop-clause*。LOOP的if和when关键字Common Lisp中同名的关键词含义不同，在这里它们是同义词——行为上没有区别。

下面这个相当傻的循环演示了几种不同形式的LOOP条件子句。函数update-analysis将在每次通过循环时被调用，其参数是条件子句中的汇聚子句最后更新的不同变量的值。

```
(loop for i from 1 to 100
      if (evenp i)
        minimize i into min-even and
        maximize i into max-even and
        unless (zerop (mod i 4))
          sum i into even-not-fours-total
        end
        and sum i into even-total
      else
        minimize i into min-odd and
        maximize i into max-odd and
        when (zerop (mod i 5))
          sum i into fives-total
        end
        and sum i into odd-total
      do (update-analysis min-even
                          max-even
                          min-odd
                          max-odd
                          even-total
                          odd-total
                          fives-total
                          even-not-fours-total))
```

22.11　设置和拆除

LOOP语言设计者早就预见到：循环在实际使用中总是以一些设置初始环境的代码开始，循

环结束后还会有更多的代码来处理由循环所计算出来的值。举一个简单的Perl例子[①]，如下所示：

```
my $evens_sum = 0;
my $odds_sum  = 0;
foreach my $i (@list_of_numbers) {
  if ($i % 2) {
    $odds_sum += $i;
  } else {
    $evens_sum += $i;
  }
}
if ($evens_sum > $odds_sum) {
  print "Sum of evens greater\n";
} else {
  print "Sum of odds greater\n";
}
```

这段代码中的循环是foreach语句。但foreach本身并不能独立工作：循环体中的代码引用了循环开始前的两行代码中声明的变量。[②]而循环所做的所有工作假如果没有了后面那个if语句的话也就毫无意义了，if语句在循环结束后输出结果。在Common Lisp中，**LOOP**结构也是一个可以返回值的表达式，因此通常更需要做的一件事是在循环结束之后生成一个有用返回值。

所以，**LOOP**的设计者说，应该提供一种方式将原本应该放在循环中的那些代码也塞进循环里。这样，**LOOP**就提供了两个关键字，initially和finally，用于引入那些原本会运行在循环主体以外的代码。

在initially或finally之后，这些子句由所有需要在下一个循环子句开始之前或者循环结束之后运行的多个Lisp形式所组成。所有的initially形式会被组合一个的“序言”，在所有局部循环变量被初始化以后和循环体开始之前运行一次。所有的finally形式则被简单地组合成一个“尾声”，在循环体的最后一次迭代结束以后运行。序言和尾声部分的代码都可以引用局部循环变量。

就算循环迭代了零次，序言部分也总是会运行。但是循环有可能在下列任何情况发生时不会运行尾声：

- 执行了一个return子句。
- **RETURN**、**RETURN-FROM**或其他控制构造的传递操作在循环体中的一个Lisp形式中被调用[③]。
- 循环被一个always、never或thereis子句终止，我将在下一节里讨论这种情况。

在尾声部分的代码中，**RETURN**或**RETURN-FROM**可被用来显式提供一个循环的返回值。这个显式的返回值将比其他汇聚或终止测试子句所提供的值具有更高的优先级。

另外，为了使**RETURN-FROM**从一个特定的循环中返回（这在嵌套的**LOOP**表达式中是有用的），

① 我并不是故意选择Perl的，这个例子在任何语法基于C的语言里看起来都差不多。

② 在Perl里，如果你没有使用use strict的话，Perl会允许你随意使用未经声明的变量。但你应当总是在Perl中使用use strict。Python、Java或C中的等价代码将总是会要求声明变量。

③ 你可以使用局部宏**LOOP-FINISH**让整个循环从循环体中的某段Lisp代码中直接正常返回，同时有机会执行循环的尾声部分。

你可以使用循环关键字named为**LOOP**命名。如果一个named子句出现在一个循环中，那么它必须是第一个子句。举一个简单的例子，假设lists是一个列表的列表，而你想要在这些嵌套的列表中查到匹配某些特征的项。你可以像下面这样使用一对嵌套的循环来找到它：

```
(loop named outer for list in lists do
      (loop for item in list do
            (if (what-i-am-looking-for-p item)
              (return-from outer item))))
```

22.12　终止测试

尽管for和repeat子句提供了控制循环次数的方法，但有时你需要更早地中断循环。你已经知道do子句里的return子句、**RETURN**或**RETURN-FROM**形式可以立即终止循环。但正如存在一些用来汇聚值的通用模式那样，也存在用来决定何时终止循环的通用模式。在**LOOP**中这些模式是由终止子句while、until、always、never和thereis来提供的。它们全都遵循相同的模式：

loop-keyword test-form

五种子句都会在每次通过迭代时对*test-form*求值，然后基于得到的值来决定是否终止循环。它们的区别在于，如果终止循环的话需要什么条件以及如何决定。

循环关键字while和until代表了“温和的”终止子句。当它们决定终止循环时，控制会传递到尾声部分，并跳过循环体的其余部分。尾声部分随后会返回一个值或是做任何想做的事情来结束循环。while子句在测试形式首次为假时终止循环；相反地，until子句在测试形式首次为真时停止循环。

另一个温和的终止形式是由**LOOP-FINISH**宏所提供的。这是一个正规的Lisp形式，并非一个循环子句，因此它可以用在一个do子句的Lisp形式中的任何地方。它也会导致立即跳转到循环的尾声部分。这在是否跳出循环的判断难以用一个简单的while或until子句来表达时是有用的。

另外三个子句即always、never和thereis，采用极端偏执的方式来终止循环。它们立即从循环中返回，不但跳过任何连续的循环子句而且还跳过尾声部分。它们还为整个循环提供了默认的返回值，哪怕是它们没有导致循环终止。尽管如此，如果循环不是因为这些终止测试中的一个而终止的话，那么尾声部分还是有机会运行，并返回一个值以代替终止子句所提供的默认值的。

由于这些子句提供了它们自己的返回值，因此它们不能跟汇聚类子句配合使用，除非汇聚子句带有一个into下级子句。编译器（或解释器）应当在编译期报告此类错误。always和never子句仅返回布尔值，因此在你需要用一个循环表达式来构成谓词时，它们将是最有用的。你可以使用always来确认循环的每次迭代过程中测试形式均为真。相反地，never测试每次迭代中测试形式均为假。如果测试形式失败了（在always子句中返回**NIL**或是在never子句中返回非**NIL**），那么循环将被立即终止，并返回**NIL**。如果循环可以一直运行直到完成，那么就会提供默认值**T**。

举个例子，如果你想要测试一个列表numbers中的所有数都是偶数，可以写成这样：

```
(if (loop for n in numbers always (evenp n))
    (print "All numbers even."))
```

下面是等价的另一种写法：

```
(if (loop for n in numbers never (oddp n))
    (print "All numbers even."))
```

thereis子句被用来测试是否测试形式“曾经”为真。一旦测试形式返回了一个非NIL的值，那么循环就会终止并返回该值。如果循环得以运行到完成，那么thereis子句会提供默认值NIL。

```
(loop for char across "abc123" thereis (digit-char-p char)) → 1

(loop for char across "abcdef" thereis (digit-char-p char)) → NIL
```

22.13 小结

现在你已经看到了LOOP功能的所有主要特性。只要你遵循下列规则就可以将我所讨论过的任何子句组合在一起：

- 如果有named子句的话，它必须是第一个子句。
- 在named子句后面是所有的initially、with、for和repeat子句。
- 然后是主体子句：有条件和无条件的执行、汇聚和终止测试。[1]
- 以任何finally子句结束。

LOOP宏将展开成完成下列操作的代码：

- 初始化所有由with或for子句声明的局部变量，以及由汇聚子句创建的隐含局部变量。提供初始值的形式按照它们在循环中出现的顺序进行求值。
- 执行由任何initially子句（序言部分）所提供的形式，以它们出现在循环中的顺序来执行。
- 迭代，同时按照下面一段文字所描述的过程来执行循环体的代码。
- 执行由任何finally子句（尾声部分）所提供的形式，以它们出现在循环中的顺序来执行。

当循环在迭代时，循环体被执行的方式是首先步进那些迭代控制变量，然后以出现在循环中的顺序执行任何有条件或无条件的执行、汇聚或终止测试子句。如果循环中的任何子句终止了循环，那么循环体的其余部分将被跳过，然后整个循环可能在运行了尾声部分以后返回。

这基本上就是所有的内容了。[2]本书后面的代码中将频繁用到LOOP，因此有必要对它多些了解。除此之外，用不用它就完全取决于你了。

有了这些基础，就可以进入本书其余部分的实践性章节了。首先是编写一个垃圾过滤器。

---

① 一些Common Lisp实现允许你交替使用主体子句和for子句，但这在严格来讲是未定义的，并且另一些实现会拒绝这样的循环。

② 关于LOOP，我尚未讨论过的一个方面是用来声明循环变量类型的语法。当然，我也还没有讨论过LOOP之外的类型声明。我将在第32章里谈及这个一般主题。对于它们与LOOP配合使用的细节，请参考你所喜爱的Common Lisp手册。