第5章 更复杂的函数

在这一章中，在前面几章已经学习过的内容的基础上考察几个更复杂的函数。copy-to-buffer 函数展示了在一个函数定义中使用两次save-excursion表达式的情况，而 insert-buffer 函数展示了在一个 interactive 表达式中使用 *以及使用 or 函数，并且还将展示一个名字和这个名字所指的对象两者之间的重要区别。

5.1 copy-to-buffer 函数的定义

理解了 append-to-buffer 函数是如何工作的之后，copy-to-buffer 函数就很容易理解了。这个函数将文本拷贝进一个缓冲区，它不是追加到原有缓冲区，而是替换了原有缓冲区中的文本。copy-to-buffer 函数的代码几乎与append-to-buffer 函数的代码完全一样，不同之处仅在于它使用了 erase-buffer 函数，并两次使用了 save-excursion 函数。(关于 append-to-buffer 的描述参见4.4节，“append-to-buffer 函数的定义”。)

copy-to-buffer 函数体如下所示：

```
...
(interactive "BCopy to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (save-excursion
      (set-buffer (get-buffer-create buffer))
      (erase-buffer)
      (save-excursion
        (insert-buffer-substring oldbuf start end)))))
```

这些代码与 append-to-buffer 函数定义中的代码类似：仅在切换到要拷贝文本的缓冲区之后的那部分代码与 append-buffer 函数代码开始不同—— copy-to-buffer 函数删除了缓冲区中原有的内容。(这就是“取代”一词的含义；取代文本，对 Emacs 而言就是删除原有的内容后插入新的文本。)在删除缓冲区的原有内容之后，save-excursion 第二次被使用，新的文本被插入进来。

为什么要两次使用 save-excursion 函数呢？再一次考查这个函数做些什么。

在结构上说，copy-to-buffer 函数体如下所示：

```
(let (bind-oldbuf-to-value-of-current-buffer)
  (save-excursion              ; First use of save-excursion.
    change-buffer
      (erase-buffer)
      (save-excursion          ; Second use of save-excursion.
        insert-substring-from-oldbuf-into-buffer)))
```

第一次使用 save-excursion 函数使 Emacs 返回到将要拷贝文本的缓冲区。这是清楚的，

就像在 append-to-buffer 函数中使用的那样。为什么要第二次使用save-excursion函数呢? 原因在于 insert-buffer-substring 函数总是在被插入内容的缓冲区的域的末尾设置位点。函数定义中的第二个 save-excursion 函数使Emacs 在被插入内容的缓冲区的域的开始设置位点。在绝大多数情况下，用户更希望位点在被插入文本的开始处。(当然，copy-to-buffer 函数返回到原来的缓冲区——但是如果用户切换到文本插入的缓冲区，位点将处于文本的开始处。因而，第二次使用这个 save-excursion 函数是不错的。)

5.2 insert-buffer 函数的定义

insert-buffer 函数是另外一个与缓冲区有关的函数。这个命令将另外一个缓冲区内容拷贝到当前缓冲区中。它与 append-to-buffer 或者 copy-to-buffer 函数正好相反，因为它们是从当前缓冲区中拷贝一个域内的文本到另外一个缓冲区。

除此之外，这个函数的代码显示了如何在 interactive 表达式中使用一个缓冲区。这个缓冲区可能是只读的。同时这个函数展示了一个对象的名字与这个对象本身实际所指的内容之间的重要区别。下面就是这个函数的代码：

```
(defun insert-buffer (buffer)
  "Insert after point the contents of BUFFER.
Puts mark after the inserted text.
BUFFER may be a buffer or a buffer name."
  (interactive "*bInsert buffer: ")
  (or (bufferp buffer)
      (setq buffer (get-buffer buffer)))
  (let (start end newmark)
    (save-excursion
      (save-excursion
        (set-buffer buffer)
        (setq start (point-min) end (point-max)))
      (insert-buffer-substring buffer start end)
      (setq newmark (point)))
    (push-mark newmark)))
```

就像其他的函数定义一样，可以使用函数定义的模板来分析这个函数的框架：

```
(defun insert-buffer (buffer)
  "documentation..."
  (interactive "*bInsert buffer: ")
  body...)
```

5.2.1 insert-buffer 函数中的交互表达式

在insert-buffer函数中，给 interactive 表达式说明的参量有两个部分：一部分是一个星号“*”，另一部分是 “bInsert buffer:”。

1. 只读缓冲区

星号“*” 用于缓冲区是一个只读缓冲区的情况，只读缓冲区就是指一个不能改变内容的缓

冲区。如果 insert-buffer 被一个只读缓冲区调用，一条消息将打印在回显区，终端将发出蜂鸣或者闪亮一下，警告不允许往这个缓冲区插入任何东西。星号这个控制符无需后接一个换行符来分隔不同的参量。

2. 交互表达式中的 "b"

在交互表达式中的下一个参量以 "b" 字符开始。(这一点与 append-to-buffer 函数的代码不同，在那个函数定义中使用一个大写的 "B"。参见4.4节，"append-to-buffer 函数的定义"。)小写的 "b" 告诉 Lisp 解释器传送给 insert-buffer 函数的参量应是一个存在的缓冲区或者这个缓冲区的名字。(大写的 "B" 可以允许参量传送不存在的缓冲区)。Emacs 将提示你输入缓冲区的名字，并为你提供一个默认的缓冲区。如果这个缓冲区不存在，你将收到一条消息告之 "找不到该缓冲区"，你的终端也将对你发生蜂鸣叫声。

5.2.2 insert-buffer 函数体

insert-buffer 函数体有两个主要的部分：一个 or 表达式和一个 let 表达式。or 表达式的目的是为了确保 buffer 参量真正与一个缓冲区绑定在一起，而不是绑定到缓冲区的名字。let 表达式包含了将另外一个缓冲区的内容拷贝到当前缓冲区所需的代码。

从结构上说，适合insert-buffer 函数的两个表达式如下所示：

```
(defun insert-buffer (buffer)
  "documentation..."
  (interactive "*bInsert buffer: ")
  (or ...
      ...
  (let (varlist)
       body-of-let... )
```

为了理解 or 表达式如何确保参量 buffer 确实是绑定到一个缓冲区，而不是绑定到一个缓冲区的名字，首先需要理解 or 函数本身。

在讲述 or 函数之前，让我用 if 表达式重新改写函数的这一部分，这样你就可以看到这是如何以另外一种方式完成类似的工作的。

5.2.3 用 if 表达式(而不是or表达式)编写的 insert-buffer 函数

用 if 表达式来改写这个部分代码，是为了确保 buffer 的值是缓冲区本身而不是缓冲区的名字。如果 buffer 的值是缓冲区的名字，则一定要设法得到缓冲区本身。

可以想象，好比在一个会议上，一个引座员手拿一本印有你大名的花名册寻找你：引座员绑定到你的名字，而不是你，但是当引座员找到你并拉住你的手的时候，引座员就绑定到你了。

在 Lisp 中，可以这样描述这种情况：

```
(if (not (holding-on-to-guest))
    (find-and-take-arm-of-guest))
```

我们希望对一个缓冲区做同样的事情——如果没有获得缓冲区本身，就要设法得到它。

一个名为 bufferp 的谓词会告诉我们是否得到一个缓冲区本身(而不是缓冲区的名字)，

因此可以这样编写代码：

```
(if (not (bufferp buffer))              ; if-part
    (setq buffer (get-buffer buffer)))  ; then-part
```

这里，这个 if 表达式中的真假测试是 (not (bufferp buffer))，它的then部是表达式 (setq buffer (get-buffer buffer))。

在这个测试中，如果其参量是一个缓冲区， bufferp 函数返回值为“真”，但是如果其参量是一个缓冲区的名字，则函数 bufferp 的返回值为“假”。(bufferp 函数名的最后一个字符是“p”；就像前面看到的，这样使用“p”是一种习惯，它表示这个函数是一个谓词，也就是这个函数将判断一些特性的真假。参见1.8.4节，“用一个错误类型的数据对象作为参量”。)

not 函数在表达式 (bufferp buffer) 之前，因此真假测试就是：

```
(not (bufferp buffer))
```

not 函数的功能是：如果其参量为“假”，则其返回“真”；如果其参量值为“真”，则其返回“假”。因此，如果(bufferp buffer)表达式返回“真”，则 not 表达式返回“假”；反之亦然：“不真”就是“假”，“不假”就是“真”。

采用了这个真假测试后，if 表达式便这样工作：当变量 buffer 的值确实是一个缓冲区而不是它的名字时，这个真假测试返回“假”，if 表达式不对 then 部求值。这正是我们需要的，因为如果变量 buffer 的值真的是一个缓冲区，那么确实无需对它再做什么。

另一方面，当 buffer 的值不是缓冲区本身而是缓冲区的名字时，真假测试返回“真”，因而then部被求值。在这个例子中，if 表达式的then部是 (setq buffer (get-buffer buffer))。这个表达式用 get-buffer 函数通过缓冲区的名字来获得这个实际的缓冲区。setq 则将缓冲区本身的值设置给变量 buffer，以取代它的原先值(即这个缓冲区的名字)。

5.2.4 函数体中的 or 表达式

insert-buffer 函数中的 or 表达式的目的是确保 buffer 参量绑定到一个缓冲区而不仅仅是一个缓冲区的名字。前面一节介绍了这可以用 if 表达式完成。然而，insert-buffer 函数实际上使用了 or 表达式。为了理解这一点，有必要弄懂 or 表达式是如何工作的。

一个 or 函数可以有很多参量。它逐一对每一个参量求值并返回第一个其值不是nil的参量的值。同样，这是 or 表达式的一个重要特性，一旦遇到其值不是 nil 的参量之后，or 表达式就不再对后续的参量求值。

or 表达式如下所示：

```
(or (bufferp buffer)
    (setq buffer (get-buffer buffer)))
```

or 表达式的第一个参量是(bufferp buffer)。如果变量 buffer确实对应着一个缓冲区，则这个表达式返回“真”(一个非空值，not-nil)；如果 buffer 仅仅是一个缓冲区的名字，则这个表达式返回“假”。在这个 or 表达式中，如果确实是这么一回事，则or 表达式返回这个“真”值，而且不再对其下一个表达式求值——这正是我们需要的，如果 buffer 确实是一个缓

冲区，无需对 buffer 的值做任何事情。

另一个方面，如果（bufferp buffer）的值是nil，Lisp 解释器则对 or表达式的下一个元素求值。这种情况当buffer的值是一个缓冲区的名字时就会发生。下一个元素就是表达式（setq buffer (get-buffer buffer)）。这个表达式返回一个非空值，其返回值就是变量 buffer 设置的值——也就是缓冲区本身，而不是缓冲区的名字。

所有这些代码的执行结果是：符号 buffer 总是绑定到一个缓冲区本身面不是缓冲区的名字。之所以需要这些代码，是因为后面的 set-buffer 函数仅仅对一个缓冲区起作用，面不对缓冲区的名字起作用。

顺便提一下，用 or 函数，那么引座员的情况可以写成下面的形式：

```
(or (holding-on-to-guest) (find-and-take-arm-of-guest))
```

5.2.5 insert-buffer 函数中的let表达式

在确保变量 buffer 是指向一个缓冲区而不是一个缓冲区的名字之后，insert-buffer 函数继续使用 let 表达式。这定义了三个变量：start、end 和 newmark，并将它们绑定到初始值 nil。这些变量在 let 表达式内部使用，并暂时屏蔽 Emacs中所有同名的变量直到 let 表达式结束。

let 表达式的主体包含了两个 save-excursion 表达式。首先，将详细分析内层的那个 save-excursion 表达式。这个表达式如下所示：

```
(save-excursion
  (set-buffer buffer)
  (setq start (point-min) end (point-max)))
```

表达式 (set-buffer buffer) 将 Emacs 的注意力从当前的缓冲区切换到要从中拷贝文本的缓冲区。在那个缓冲区中，用 point-min 和 point-max 函数将变量 start 和 end 设置成该缓冲区的开始处和结束处。注意，这里已经演示了 setq 如何在一个表达式中给两个变量赋值。setq 的第二个参量的值被赋给第一个参量，第四个参量的值被赋给第三个参量。

内层的 save-excursion 表达式被求值之后，它恢复原来的缓冲区，但是 start 和 end 变量仍然被设置成从中拷贝文本的缓冲区的开始处和结束处。

外层的 save-excursion表达式如下所示：

```
(save-excursion
  (inner-save-excursion-expression
     (go-to-new-buffer-and-set-start-and-end)
  (insert-buffer-substring buffer start end)
  (setq newmark (point)))
```

insert-buffer-substring 函数将文本从由 buffer 中的 start 和 end 变量界定的区域拷贝到当前缓冲区。因为第二个缓冲区的全部内容都在 start 和 end 之间，因此第二个缓冲区的所有内容都被拷贝到你正在编辑的当前缓冲区。接下来，位于插入文本末尾的位点的值记录在变量 newmark 中。

外层的 save-excursion 被求值之后，位点和标记被重新定位到它们原来的位置。

然而，习惯上，标记的位置一般在新插入的文本之后，而位点的位置在新插入的文本的开始处。newmark 变量记录新插入的文本的末尾。在 let 表达式的最后一行，(push-mark newmark)表达式在这个位置设置了一个标记。(上一个标记的位置仍然可以找到，它记录在标记环中，可以键入 C-u C-SPC 返回到原来标记处。)同时，位点设置在插入的文本的开始处，也就是位点设置在你调用插入函数之前所处的位置。

整个 let 表达式如下所示：

```
(let (start end newmark)
  (save-excursion
    (save-excursion
      (set-buffer buffer)
      (setq start (point-min) end (point-max)))
    (insert-buffer-substring buffer start end)
    (setq newmark (point)))
  (push-mark newmark))
```

就像 append-to-buffer 函数那样，insert-buffer 函数使用了 let、save-excursion 和 set-buffer 函数。除此之外，这个函数展示了使用 or 函数的一个方法。在后面将一次又一次看到这些函数，它们都是构建 Lisp 程序的基本函数。

5.3 beginning-of-buffer 函数的完整定义

在前面已经讨论过beginning-of-buffer 函数的基本结构。(参见4.2节，“简化的 beginning-of-buffer 函数定义”。)本节描述这个函数定义的复杂部分。

就像前面描述的，不带参量激活 beginning-of-buffer 函数时，它将光标移动到缓冲区的开始处，并在原来光标的位置设置一个标记。然而，当带参量激活 beginning-of-buffer 函数时，如果参量是介于 1 和 10 之间的一个数，则该函数认为那个数是指缓冲区长度的十分之几，而且 Emacs 将光标移动到从缓冲区开始到这个分数值所指示的位置。因此，可以用键序列 M-< 调用这个函数，这是指将光标移动到缓冲区的开始处，也可以用这样的命令调用这个函数：C-u 7 M-<，这是指将光标移动到从缓冲区开始的这个缓冲区的70%处。如果这个作为参量的数大于 10，函数则将光标移动到缓冲区的末尾。

beginning-of-buffer 函数可以带参量调用，也可以不带参量调用，参量的使用是可选的。

5.3.1 可选参量

除非已经声明，否则 Lisp 总是希望一个函数定义中带参量的函数在被调用时要传递一个值给该参量。如果没有传递相应的值，函数就会出错，并得到这样一个错误消息：“Wrong number of arguments”。

然而，可选参量是 Lisp 的一个特性：有一个关键词可以用于告诉 Lisp 解释器某个参量是可选的。这个关键词是 &optional（在单词“optinal”之前的符号“&”是关键词的一部分）。

在一个函数定义中，如果一个参量跟在 &optinal 这个关键词后面，则当调用这个函数时就不一定要传送一个值给这个参量。

因此，beginning-of-buffer 函数定义的第一行就变成如下所示的形式：

```
(defun beginning-of-buffer (&optional arg)
```

从结构上说，整个函数如下所示：

```
(defun beginning-of-buffer (&optional arg)
  "documentation..."
  (interactive "P")
  (push-mark)
  (goto-char
    (if-there-is-an-argument
        figure-out-where-to-go
      else-go-to
      (point-min))))
```

除了这个函数在 interactive 表达式中使用了“P”参量以及将 goto-char 函数用在一个条件表达式中以判断将光标移动到何处之外，上面这个函数与simplified-beginning-of-buffer 函数很相似。

interactive 表达式中的“P”参量告诉 Emacs，如果有参量的话，就传递一个前缀参量给这个函数。一个前缀参量由键入 META 键以及后接的一个数组成的，或者由键入 C-u 和一个后接的数组成（如果你没有键入一个数，C-u默认为4）。

上面的函数定义中，if 条件表达式的真假测试很简单：它只是参量 arg 而已。如果参量 arg 有一个非空（nil）值，即当 beginning-of-buffer 函数带参量调用时，真假测试返回“真”，并且 if 表达式中的 then 部被求值；另一方面，如果不带参量调用 beginning-of-buffer 函数，这个 arg 参量为 nil，并且if 表达式的 else 部被求值。if 表达式的 else 部只是 point-min，并且当这就是 if 表达式的结果时，整个 goto-char 表达式就是 (goto-char (point-min))，这就是我们在前面看到的 beginning-of-buffer 函数的简化形式。

5.3.2 带参量的 beginning-of-buffer 函数

当带参量调用 beginning-of-buffer 函数时，就要有一个表达式计算应该传递什么值给 goto-char 表达式。这个表达式初看起来似乎相当复杂，它包含一个 if 表达式和许多算术计算。这个表达式如下所示：

```
(if (> (buffer-size) 10000)
    ;; Avoid overflow for large buffer sizes!
    (* (prefix-numeric-value arg) (/ (buffer-size) 10))
  (/
   (+ 10
      (*
       (buffer-size) (prefix-numeric-value arg))) 10))
```

就像其他复杂的表达式一样，这个表达式也可以用模板来一一揭开其中的奥秘。在这个例子中，模板就是 if 表达式模板。当用结构框架来看这个表达式时，这个表达式如下所示：

```
(if (buffer-is-large
     divide-buffer-size-by-10-and-multiply-by-arg
  else-use-alternate-calculation
```

在这个内层的 if表达式中，真假测试用于检查缓冲区的大小。之所以要检查缓冲区的大小是因为第18版的Emacs Lisp使用了不大于 8 000 000 的数字来描述缓冲区的大小（更大的数就不需要了），并当在后续的计算中遇到很大的缓冲区时，Emacs 就试图使用超大的数来描述它。在注释中提到的术语“overflow”(溢出)是指所用的数太大了。

这里有两种情况：缓冲区很大或者并不大。

1. 大缓冲区的情况

在 beginning-of-buffer 函数中，内层的 if 表达式判断缓冲区是否大于10 000 个字符。它使用 > 函数和 buffer-size 函数来完成这一工作：

```
(if (> (buffer-size) 10000)
```

当缓冲区大于 10 000 时，if 表达式的 then 部被求值。其中 then 部如下所示：

```
(*
  (prefix-numeric-value arg)
  (/ (buffer-size) 10))
```

这个表达式是一个乘法，*函数有两个参量。

其中的第一个参量是 (prefix-numeric-value arg)。当在 interactive 表达式中使用“P”参量时，作为函数参量传给函数的值是以一个“未加工的前缀参量”(raw prefix argument) 的形式传递的。(它是在一个列表中的一个数。）为了执行算术运算，有必要对它进行变换，prefix-numeric-value 函数就是完成这一工作的。

其中的第二个参量是 (/ (buffer-size) 10)。这个表达式将缓冲区的大小（数字）除以 10。这个表达式产生一个数，这个数就是指缓冲区大小的十分之一有多少字符。（在 Lisp 中，/用于除法，就像 * 用于乘法一样）。

在乘法表达式中，这个数作为一个整体乘以前缀参量的值，乘法的结构如下所示：

```
(* numeric-value-of-prefix-arg
   number-of-characters-in-one-tenth-of-the-buffer)
```

例如，如果前缀参量是“7”，则缓冲区的十分之一的值乘以 7 得到缓冲区的 70%。

如果是大缓冲区，则所有这些代码的最后结果就使 goto-char 表达式变成这样：

```
(goto-char (* (prefix-numeric-value arg)
              (/ (buffer-size) 10)))
```

这个表达式的功能是将光标置于我们需要的地方。

2. 小缓冲区的情况

如果缓冲区中包含的字符数少于 10 000 个，就要执行一个稍微不同的计算。你可能认为这不必要，因为前面的计算可以完成这个工作。然而，在一个小缓冲区中，第一种方法无法精确地将光标置于所需的那一行。这第二种方法可以更好地做到这一点。

这部分代码是：

```
(/ (+ 10 (* (buffer-size) (prefix-numeric-value arg))) 10))
```

这个函数代码看似有些复杂，但是通过逐一分析函数是如何嵌入到括号中，就可以清楚地分析出最后的结果。如果以缩进的方式重写每一个表达式，就可以更容易地阅读它。

```
(/
 (+ 10
    (*
     (buffer-size)
     (prefix-numeric-value arg)))
 10))
```

检查这些括号，会发现最内层的操作是 (prefix-numeric-value arg)，即把未加工的前缀参量转换成一个数。这个数在下面的表达式中乘以缓冲区的大小:

```
(* (buffer-size) (prefix-numeric-value arg)
```

这个乘法的结果是产生一个数，这个数可能大于缓冲区的大小——如果参量是 7，就是缓冲区的 7 倍。然后这个数再加 10，最后用这个结果除以 10，这样产生的数比缓冲区中相应比例仅仅多一个字符。

所有这些代码执行后产生的最终的一个数被传递到 goto-char 函数，并且光标就移动到那个位点。

5.3.3 完整的 beginning-of-buffer 函数

下面是 beginning-of-buffer 函数的完整形式:

```
(defun beginning-of-buffer (&optional arg)
  "Move point to the beginning of the buffer;
leave mark at previous position.
With arg N, put point N/10 of the way
from the true beginning.
Don't use this in Lisp programs!
\(goto-char (point-min)) is faster
and does not set the mark."
  (interactive "P")
  (push-mark)
  (goto-char
   (if arg
       (if (> (buffer-size) 10000)
           ;; Avoid overflow for large buffer sizes!
           (* (prefix-numeric-value arg)
              (/ (buffer-size) 10))
         (/ (+ 10 (* (buffer-size)
                     (prefix-numeric-value arg)))
            10))
     (point-min)))
  (if arg (forward-line 1)))
```

除了两个小点外，前面的讨论展示了这个函数是如何工作的。第一点处理文档字符串中的细节，第二点关于函数的最后一行。

在文档字符串中，提到了这样一个表达式：

```
\(goto-char (point-min))
```

其中的第一个括号之前有一个“\”符号。这个符号告诉 Lisp 解释器将这个表达式作为文档打印出来，而不作为一个符号表达式对它求值。

最后，当这个函数带参量调用时，beginning-of-buffer 函数的最后一行是让光标移动到后续一行的开始处：

```
(if arg (forward-line 1))
```

这个命令将光标置于缓冲区中相应于前缀参量值的位置的后续第一行的行首。这是一个好主意，它意味着光标总是置于缓冲区中至少是需要的位置。我们当然希望光标精确地置于需要到达的位置，但是这并不是必须的。如果没有精确地置于需要到达的位置，也不要抱怨太多。

5.4 回顾

下面是这一章中讨论的部分主题的一个简要小结：

- or

逐一对每一个参量求值，并返回第一个非空值（不是 nil）。如果所有参量的值都是 nil，就返回 nil。简要地说，它返回参量的第一个“真”值；如果一个参量或者其他任何参量的值为“真”时，则返回“真”值。

- and

逐一对每一个参量求值，如果有任何一个参量的值为 nil，则返回 nil。如果没有 nil 值，则返回最后一个参量的值。简要地说，仅当所有参量都是“真”值时，它才返回一个“真”值；如果一个参量和其他所有参量的值都是“真”值时，则返回“真”值。

- &optional

在函数定义中用于指出一个参量是可选参量。这意味着这个函数可以带参量调用，也可以不带参量调用。

- prefix-numeric-value

将一个由 (interactive "P") 产生的未加工的前缀参量转换成一个数值。

- forward-line

将光标移动到下一行的行首，如果其参量大于 1，则移动多行。如果无法移动所需的行数，forward-line 就移动尽可能多的行数，并返回它实际少移动的行数。

- erase-buffer

删除当前缓冲区的全部内容。

- bufferp

如果其参量是一个缓冲区则返回“真”，否则返回“假”(nil)。

5.5 &optional 参量练习

编写一个带可选参量的交互函数，这个函数要测试函数被调用时是否有参量（其值是一个数），这个数是否大于或小于 fill-column 的值，并将结果以一个消息的形式给出。然而，如果不带参量调用这个函数时，则使用 56 作为默认值。