第4章 与缓冲区有关的函数

在这一章，我们将详细学习几个在 GNU Emacs 中使用的函数。我们称之为一次“浏览”。这些函数是作为 Lisp 代码的例子使用的，但是这些并不是很具创造性的例子。除了其中第一个简化了的函数定义之外，这些函数显示了在 GNU Emacs 中实际使用的代码。你可以从这些函数定义中学到很多东西。这里描述的这些函数都与缓冲区有关。在后面，我们将学习其他一些函数。

4.1 查找更多的信息

在这个浏览中，我将对要遇到的每一个函数进行或者详细或者简单的描述。如果你感兴趣的话，可以键入 C-h f 以及函数名（当然还要加上回车键RET），随时得到任何一个 Emacs Lisp 函数的全部文档。类似地，可以键入 C-h v 和变量名（以及回车键RET）得到任何变量的全部文档。

同样，如果要在一个原始的源代码文件中查看一个函数定义，可以使用find-tags 函数跳到相应的位置。键入 M-.（即同时键入 META 键和句点，或者键入ESC键后再键入句点），然后在提示符下输入要查看源代码的函数的函数名，例如mark-whole-buffer，然后键入回车键RET。Emacs 将切换缓冲区并在屏幕上显示函数的源代码。要切换回原来的缓冲区，键入 C-x b RET。

根据你的 Emacs 拷贝的初始缺省设置值的不同，你可能还需要定义一个“标记表”（tags table），这是一个名为“TAGS”的文件。这个文件最可能在 “emacs/src”目录中，因此应使用 M-x visit-tages-table 命令并指定一个类似“/usr/local/ lib/emacs/19.23/src/TAGS”这样的路径名。关于如何创建自己的“TAGS”文件，可以参见《*GNU Emacs* 技术手册》的“标记表”一节，也可以参见本书的12.5节，“创建自己的‘TAGS’文件”。

当你对 Emacs Lisp 更加熟悉后，你会频繁地使用 find-lags 去浏览源代码，并且可以创建自己的“TAGS”标记表。

顺便说一说，包含 Lisp 代码的文件习惯上称为*库*（*library*）。这种隐喻说法来自于特别定义的库，例如法律库、工程库面非一般的图书馆。每一个库，或者一个文件，都是与某个主题或者某项活动有关的函数。例如，“abbrev.el”用于处理缩写和其他输入快捷键，面“help.el”用于在线帮助。（有时为了某个任务需要几个库，例如各种“rmail...”文件为阅读电子邮件提供代码。）在《*GNU Emacs* 技术手册》中，将看到这样的句子：“C-h p 命令让你用主题关键字搜索 Emacs Lisp 标准库”。

4.2 简化的 beginning-of-buffer 函数定义

beginning-of-buffer 命令是一个很好的开始，因为你可能已经对它有所熟悉，并且

它易于理解。作为一个交互命令，beginning-of-buffer 函数将光标移动到缓冲区的开始位置，在原来的位置设置一个标记。这个函数一般绑定到 M-<。

在这一节中，我们将讨论这个函数的一个简化版本，这个简化版本展示如何经常使用这个函数。这个简化的函数像所编写的那样运作，但是它没有包含处理复杂选项的代码。在另一节，我们将描述这个函数的全部代码（参见5.3节，“beginning-of-buffer 函数的完整定义”）。

在考虑这个函数的代码之前，让我们看看这个函数定义应当包含哪些部分：它必须包含使之成为交互命令的 表达式，这样它才能在键入 M-x beginning-of-buffer 或键入C-<这样的键序列时被调用；它也必须包含在缓冲区中的原始位置保留标记的代码，以及将光标移动到缓冲区开始处的代码。

下面就是这个简化的 beginning-of-buffer 函数的完整代码：

```
(defun simplified-beginning-of-buffer ()
  "Move point to the beginning of the buffer;
leave mark at previous position."
  (interactive)
  (push-mark)
  (goto-char (point-min)))
```

就像所有的函数定义一样，这个定义在特殊表 defun 之后有五个部分：

1) 函数名：在这个例子中，就是 simplified-beginning-of-buffer。

2) 参量列表：在这个例子，是一个空列表（）。

3) 文档字符串。

4) 交互表达式。

5) 函数体。

在这个函数定义中，参量列表是空的。这意味着这个函数无需任何参量。（当讨论这个函数的完整定义时，将看到它可能被传递一个可选参量。）

交互表达式（interactive）告诉 Emacs 这个函数可以被交互地使用。在这个例子中，interactive 没有参量，因为 simplified-beginning-of-buffer 不需要参量。

函数体由两行组成：

```
(push-mark)
(goto-char (point-min))
```

上述两行中的第一行是一个表达式（push-mark）。当 Lisp 解释器对这个表达式求值时，它在光标的当前位置（无论是在哪个位置）设置一个标记。这个标记的位置被保存到标记环中。

第二行是表达式（goto-char (point-min)）。这个表达式将光标跳到本缓冲区的最小可能位点处，也就是缓冲区的开始处（如果变窄开启，就是这个缓冲区中可访问部分的开始处。有关内容参见第6章“变窄和增宽”。

push-mark 命令将一个标记设置在光标移动前所处的位置。光标将根据(goto-char (point-min))表达式的要求移动到缓冲区的开始处。接下来，如果你愿意的话，你可以返回到原来所处的位置，只需键入 C-x C-x 即可。

这就是这个简化函数的完整定义！

当在阅读代码（如这个例子）的过程中遇到不熟悉的函数(如goto-char 函数)时，可以用describe-function 命令找到关于这些函数功能的说明。键入 C-h f 并随后输入函数名和回车键RET，就可以使用这个命令。describe-function命令将在一个“*Help*”窗口中打印函数的文档字符串。例如，goto-char 函数的文档如下所示：

```
One arg, a number.  Set point to that number.
Beginning of buffer is position (point-min),
end is (point-max).
```

(describe-function 命令的提示符将自动提供光标前的那个符号，因此你能够将光标移动到那个不熟悉的函数之后并随后键入 C-h f RET即可得到相应函数的说明，从而可以节省输入的时间)。

end-of-buffer 函数定义就是用类似于 beginning-of-buffer 函数定义的方式编写的，不同之处在于函数体用 (goto-char (point-max)) 表达式代替了（goto-char (point-min)）表达式。

4.3 mark-whole-buffer 函数的定义

mark-whole-buffer 函数并不比 simplified-beginning-of-buffer函数更难理解。然而，在这个例子中，将讨论这个函数的完整代码而不是一个简化的版本。

虽然 mark-whole-buffer 函数并不像 beginning-of-buffer 函数那样被经常使用，但是它依然很有用：它将整个缓冲区作为一个域来标记，方法是将位点置于缓冲区开始的位置，在缓冲区的末尾位置放一个标记。这个命令一般绑定到 C-x h。

这个函数的完整定义代码如下所示：

```
(defun mark-whole-buffer ()
  "Put point at beginning and mark at end of buffer."
  (interactive)
  (push-mark (point))
  (push-mark (point-max))
  (goto-char (point-min)))
```

就像其他函数一样，mark-whole-buffer 函数适合于用作函数定义的模板。模板如下所示：

```
(defun name-of-function (argument-list)
  "documentation..."
  (interactive-expression...)
  body...)
```

这个函数是这样工作的：函数名是 mark-whole-buffer；函数名后面跟着一个空列表“()”，这意味着这个函数无需参量。函数文档就跟在后面。

再下面一行是（interactive）表达式，这个表达式告诉 Emacs 该函数可以被交互地使用。这些细节都与前一节的 simplified-beginning-of-buffer 函数类似。

mark-whole-buffer 的函数体

mark-whole-buffer 函数的函数体由三行组成：

```
(push-mark (point))
(push-mark (point-max))
(goto-char (point-min))
```

其中第一行是表达式（(push-mark (point)）。

这一行的作用与 simplified-beginning-of-buffer 函数的函数体的第一行（push-mark）的作用完全一样。这两个例子中，Lisp 解释器都在当前光标所在的位置设置一个标记。

我不知道为什么在 mark-whole-buffer 函数中这个表达式写成(push-mark (point))这种形式，而在 beginning-of-buffer 函数中这个表达式却写成（push-mark）形式。可能是编写这个代码的人不知道 push-mark 函数的参量是可选的，而且如果没有传送参量给 push-mark，这个函数自动地在当前位点的位置设置标记。或者，这可能是为了与下一行结构一致。无论如何，这一行是使 Emacs 决定位点的位置并在此设置一个标记。

mark-whole-buffer 函数的下一行是（push-mark (point-max)）。这个表达式在缓冲区中数值最大的位点处设置一个标记。这个位点将是缓冲区的末尾（或者，如果变窄 narrowing 开启，就是缓冲区的可访问域的末尾，有关变窄的详细内容，参见第6章“变窄和增宽”）。设置好这个标记之后，原来的标记(即设置在位点上的标记)就不再是标记了，但是 Emacs 记住了它的位置，就像其他最近的标记被记住一样。这就是说，如果你乐意的话，可以通过键入C-u C-SPC 两次来返回到原来的位点处。

最后，这个函数体的最后一行是（goto-char (point-min)）。这种编写方法与 heginning-of-buffer 函数中的编写方法完全类似。这个表达式将光标移动到缓冲区中位点的最小值处，也就是缓冲区的开始处（或者是该缓冲区中可访问域的开始处）。这个函数求值的结果，就是位点被置于缓冲区的开始，标记被设置在缓冲区的末尾。因此，整个缓冲区就是一个域。

4.4 append-to-buffer函数的定义

append-to-buffer 命令几乎就像 mark-whole-buffer 命令一样简单。这个命令的功能就是从当前缓冲区中拷贝一个域（即缓冲区中介于位点和标记之间的区域）到一个指定的缓冲区。

append-to-buffer 命令使用 insert-buffer-substring 函数来拷贝一个域。由 insert-buffer-substring 函数名就可以看出这个函数的功能是：它从一个缓冲区提取一部分作为一个字符串，即“子字符串”（substring），并将这个字符串插入到另外一个缓冲区中。append-to-buffer 函数的绝大部分工作就是为 insert-buffer-substring 函数创建适当的条件：即它的代码必须指定字符串的来源缓冲区和目的缓冲区。下面就是这个函数定义的全部内容：

```
(defun append-to-buffer (buffer start end)
  "Append to specified buffer the text of the region.
```

```
It is inserted into that buffer before its point.

When calling from a program, give three arguments:
a buffer or the name of one, and two character numbers
specifying the portion of the current buffer to be copied."
  (interactive "BAppend to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (save-excursion
      (set-buffer (get-buffer-create buffer))
      (insert-buffer-substring oldbuf start end))))
```

通过观察一系列已经填充好的模板，可以逐一理解这个函数。

最外一层模板就是函数定义模板。在这个例子中，它看起来如下所示：

```
(defun append-to-buffer (buffer start end)
  "documentation..."
  (interactive "BAppend to buffer: \nr")
  body...)
```

函数定义的第一行包含了函数名以及它的三个参量。这些参量中，buffer 参量是指拷贝文本的目的缓冲区，start和 end 参量是指将要被拷贝的当前缓冲区中指定域的起始和终止位点。

函数的下一个部分就是函数文档，它是非常清楚和完整的。

4.4.1 append-to-buffer 函数的交互表达式

因为append-to-buffer函数将被交互地使用，所以函数必须有一个 interactive表达式。（有关 interactive 的评述，参见3.3节，“使函数成为交互函数”。）函数中的这个交互表达式读作：

```
(interactive "BAppend to buffer: \nr")
```

这个表达式有一个位于双引号中的参量，这个参量有两部分，其间由“\n”分隔开来。

参量的第一个部分是“BAppend to buffer:”。这里，“B”控制符告诉Emacs要求输入缓冲区名并将这个名字传送给函数。Emacs 将在小缓冲区中打印出“B”字符后面的字符串“Append to buffer:”来提示用户输入这个缓冲区名。然后，Emacs 将函数参量列表中的参量 buffer绑定到指定的缓冲区。

换行符“\n”将参量的两个部分分隔开，参量的第二部分就是“r”。它告诉 Emacs 将函数参量列表中将号“buffer”之后的两个参量（即 start 和 end）绑定到位点和标记的值上。

4.4.2 append-to-buffer 函数体

append-to-buffer 函数的函数体从 let 表达式开始。

就像我们在前面已经看到的那样（参见3.6节，“let参数”），let 表达式的目的是创建一个或者多个在 let 表达式主体中使用的变量，并对它们赋初使。这意味着，这样的变量将不会与 let 表达式之外的同名变量混淆。

通过下面显示的用let表达式的 append-to-buffer 函数的模板，我们将看到let 表达式是如何在整体上符合函数定义的要求。

```
(defun append-to-buffer (buffer start end)
  "documentation..."
  (interactive "BAppend to buffer: \nr")
  (let ((variable value))
       body...)
```

let 表达式有三个元素：

1) 符号let；

2) 一个变量列表，在这个例子中，这个变量列表包含一个两元素列表（*variable value*）

3) let 表达式主体。

在append-to-buffer函数中，变量列表如下所示：

```
(oldbuf (current-buffer))
```

在 let 表达式的这个部分，一个变量（即 oldbuf）被绑定到由（current-buffer）表达式返回的值上。这个oldbuf变量用于跟踪当前的缓冲区。

变量列表的元素是由一组括号包围起来的，因此 Lisp 解释器能够将变量列表从let 表达式主体中区分出来。结果，变量列表中的这个两元素列表被一组限制性的括号包围起来。这一行如下所示：

```
(let ((oldbuf (current-buffer)))
  ... )
```

oldbuf 前面的第一个括号标明的是变量列表的界限，而第二个括号标明的是这个两元素列表（oldbuf (current-buffer)）的开始，如果你没有意识到这一点，oldbuf 前面的这两个括号可能会使你大吃一惊。

4.4.3 append-to-buffer函数中的 save-excursion

append-to-buffer 函数中 let 表达式的主体由一个 save-excursion 表达式组成。

这个 save-excursion 函数保存位点和标记的位置，并当这个函数体中的其他表达式都被求值之后恢复位点和标记到相应位置。另外，save-excursion 保存原始的缓冲区并恢复它。这就是在 append-to-buffer 函数中如何使用 save-excursion 函数的方法。

顺便提一下，值得注意的是：一个 Lisp 函数一般都具有很好的格式，因此在多行展开中包含的所有内容比第一个符号缩进得更多。在下面这个函数定义中，let 表达式就比 defun 缩进得更多，而 save-excursion 表达式比 let 表达式又缩进更多：

```
(defun ...
  ...
  ...
  (let...
    (save-excursion
      ...
```

从这种格式化编排约定可以很容易看出 save-excursion 表达式主体中的两行由与 save-excursion 联系在一起的括号包围，就像 save-excursion 表达式本身由与 let 表达式联系在一起的括号包围一样：

```
(let ((oldbuf (current-buffer)))
  (save-excursion
    (set-buffer (get-buffer-create buffer))
    (insert-buffer-substring oldbuf start end))))
```

save-excursion 函数的使用可以被看做是填充下面这个模板的一个过程：

```
(save-excursion
  first-expression-in-body
  second-expression-in-body
  ...
  last-expression-in-body)
```

在这个函数中，save-excursion 表达式的主体仅仅包含两个表达式。这个表达式如下所示：

```
(set-buffer (get-buffer-create buffer))
(insert-buffer-substring oldbuf start end)
```

当对 append-to-buffer 函数求值时，save-excursion 主体中的两个表达式依次被求值。后一个表达式的值作为 save-excursion 函数的值被返回，而第一个表达式被求值仅仅是一个附带效果。

save-excursion 函数主体的第一行使用 set-buffer 函数来将当前缓冲区变换到另外一个指定的缓冲区。这个指定的缓冲区就是用 append-to-buffer 函数的第一个参量指出的另一个缓冲区。（改变缓冲区仅仅是它的附带效果；就像我们在前面说到的那样，在 Lisp 中，附带效果经常是我们所需要的。）第二行完成这个函数的主要工作。

set-buffer 函数将 Emacs 的注意力转移到文本将要拷贝到的目的缓冲区，面save-excursion 函数将从这个缓冲区返回。完成这一工作的一行表达式如下所示：

```
(set-buffer (get-buffer-create buffer))
```

这个列表中最内层的表达式是（get-buffer-create buffer）。这个表达式使用了 get-buffer-create 函数，get-buffer-create 函数要么获得已经命名的缓冲区，要么用给定的名字创建一个缓冲区（如果给定名字没有对应的缓冲区）。这意味着可以使用 append-to-buffer 函数将一段文本放到一个原本不存在的缓冲区中。

get-buffer-create 函数同时使 set-buffer 避免了一个不必要的错误：set-buffer 函数需要一个缓冲区；如果你指定一个实际上不存在的缓冲区给它，Emacs 将阻止你这样做。因为如果缓冲区不存在，get-buffer-create 将创建一个缓冲区，因此 set-buffer 函数总会得到一个存在的缓冲区。

append-to-buffer 的最后一行所做的工作就是增添文本：

```
(insert-buffer-substring oldbuf start end)
```

insert-buffer- substring函数从作为其第一个参量指定的缓冲区中拷贝一个字符串，并将其插入到当前的缓冲区中。在这个例子中，传送给 insert-buffer-substring 的参量是由 let 创建并绑定的变量的值，也就是 oldbuf 的值，它是当发出 append-to-buffer 命令时的当前缓冲区。

insert-buffer-substring 函数执行完之后，save-excursion 将恢复对原来缓冲区的操作，并且append-to-buffer 将完成其工作。

用一个粗略的框架来描述这个函数，其函数体的工作如下所示：

```
(let (bind-oldbuf-to-value-of-current-buffer)
  (save-excursion                          ; Keep track of buffer.
    change-buffer
    insert-substring-from-oldbuf-into-buffer)

  change-back-to-original-buffer-when-finished
let-the-local-meaning-of-oldbuf-disappear-when-finished
```

总之，append-to-buffer 函数是这样工作的：它在变量 oldbuf 中保存当前缓冲区的值；并获得一个新的缓冲区(如果需要的话就创建一个新的缓冲区)，然后使 Emacs 切换到这个缓冲区。使用 oldbuf 的值，这个函数将来自原来缓冲区的文本域插入到新的缓冲区中，然后用 save-excursion 函数返回到原来的缓冲区。

在考查这个 append-to-buffer 函数时，你已经深入接触了一个相当复杂的函数。它演示了如何使用 let 和 save-excursion，以及如何变换到其他缓冲区并返回原来的缓冲区。许多函数定义中都是这样使用 let、save-excursion 和 set-buffer的。

4.5 回顾

下面简单地小结一下本章讨论过的函数。

• describe-function、describe-variable

打印一个函数或者一个变量的文档。习惯上将它绑定到 C-h f 和 C-h v。

• find-tag

找到存放某个函数或者变量的源代码的文件，并切换到这个缓冲区，将位点（光标）置于相应函数或者变量的开始处。习惯上将它绑定到 M-。

• save-excursion

保存位点和标记的位置，并在 对save-excursion 参量求值之后恢复这些值。它也保存当前缓冲区并返回到该缓冲区。

• push-mark

在指定位置设置一个标记，并在标记环中记录原来标记的值。标记是缓冲区中的一个位置，即使有一些文本被从缓冲区删除或者增加到缓冲区，标记仍将保持它的相对位置。

• goto-char

将位点设置为由参量值指定的位置。参量值可以是一个数，也可以是一个标记，甚至可以是一个返回一个位置的数字的表达式，如（point-min）。

• insert-buffer-substring

将来自一个缓冲区（这是被作为一个参量而传递给函数的）的文本域拷贝到当前缓冲区。

• mark-whole-buffer

将整个缓冲区标记为一个域。一般将这个函数绑定到 C-x h。

• set-buffer

将 Emacs 的注意力转移到另一个缓冲区，但是不改变显示的窗口。这通常是由另外的人在不同的缓冲区中执行程序时使用。

• get-buffer-create、get-buffer

寻找一个已指定名字的缓冲区，或当指定名字的缓冲区不存在时就创建它。如果指定名字的缓冲区不存在，get-buffer 函数就返回 nil。

4.6 练习

- 编写自己的 simplified-end-of-buffer 函数定义，然后测试它是否能工作。
- 用 if 和 get-buffer 编写一个函数，这个函数要打印一个说明某个缓冲区是否存在的消息。
- 用find-tag找到 copy-to-buffer 函数的源代码。