第8章　剪切和存储文本

在 GNU Emacs 中，无论你何时用“kill”命令从一个缓冲区中剪切一段文本，它总是存储在一个列表中，而且你可以用一个“yank”命令将其重新取回来。

（在 Emacs 中使用“kill”一词来表示那些并未破坏对象值的过程实在是一个不幸的历史性事故。一个好得多的词可能是“clip”（修剪），因为这正是“kill”命令所完成的工作。它们将文本从缓冲区中剪切出来，放到存储设备中，并可以从存储设备中将其取回来。我经常试图在出现“kill”命令的所有地方都用“clip”来取代它，在出现“killed”的所有地方都用“clipped”来取代它。）

当将文本从缓冲区中剪切出来时，它被储存在一个列表中，连续的一段文本在列表中也是连续地存放的，因此这个列表看起来如下所示：

```
("a piece of text" "last piece")
```

cons 函数能用于往这个列表中增加一段文本片断，就像这样：

```
(cons "another piece"
      '("a piece of text" "last piece"))
```

如果对这个表达式求值，则三个元素的列表将会出现在回显区中：

```
("another piece" "a piece of text" "last piece")
```

使用 car和nthcdr函数，能从这个列表中将任何一个文本重新提取出来。例如，在下面的代码中，nthcdr 1... 返回由参量列表的第一个元素之外的所有元素组成的列表，car 函数则返回这个中间列表的第一个元素——也就是原始列表的第二个元素：

```
(car (nthcdr 1 '("another piece"
                 "a piece of text"
                 "last piece")))
     ⇒ "a piece of text"
```

当然，Emacs 中实际的函数比这个例子更加复杂。必须编写剪切和找回 文本的代码，以便 Emacs 能计算出列表中的那个元素是你所需要的——第一个、第二个、第三个.....。除此之外，当你到达列表末尾时，Emacs 会重新回到第一个元素，而不是空元素。

保存被剪切的一段文本的列表被称为kill 环（*kill ring*）。这一章首先介绍这个 kill 环，然后用 zap-to-char 函数这个例子来了解如何使用这个列表。zap-to-char函数调用另外一个对kill环操作的函数，因而在介绍 zap-to-char 这个函数之前首先介绍那个函数，就像在攀登高峰之前，先爬小山。

随后的一章将描述如何将被剪切的文本重新找回。参见第10章，“找回文本”。

8.1 zap-to-char函数

在 GNU Emacs 第18版和 第19版中，zap-to-char函数的代码是不同的。第19版中的实现更为简单，工作方式与第18版中的实现略有不同。我们首先看看第19版中的这个函数，然后再看看第18版中的代码。

在 Emacs 第19版中，交互的 zap-to-char 函数的功能就是：将光标当前位置（即位点）与出现特定字符的下一个位置之间这一区域中的文本剪切掉。zap-to-char 函数剪切的文本放在 kill 环中，并能通过键入 C-y(yank) 命令从 kill 环中找回这些文本。如果这个命令带有参量，它就将从位点处到特定字符出现了参量所示次数的那个位置之间这一区域内的文本剪切掉。因而，如果光标在这个句子 [⊖] 的开头，而指定字符是“s”，则“Thus”一词将被剪切。如果给定的参量是2，“Thus, if the curs”将被剪切，即从位点到第二次出现指定字符“s”之间的文本（包含“cursor”中的指定字符 “s”）被剪切了。

Emacs 第18版中，这个函数将位点到指定字符区域之间的文本（但不包含指定字符）剪切。因而，在上面的例子中，字符“s”就不被剪切。

除此之外，在第18版中，如果没有找到指定字符，则将位点直到缓冲区末尾整个区域内的文本全部剪切；但是在第19版中，如果发生这种情况则仅仅产生一个错误消息（不剪切任何文本）。

为了决定究竟有多少文本被剪切，两个版本的 zap-to-char 函数都使用了一个查询函数。查询函数在操纵文本的代码中广泛使用，关注查询函数与关注删除命令一样，都是值得的。

这是第19版中 zap-to-char 函数的完整代码：

```
(defun zap-to-char (arg char)  ; version 19 implementation
  "Kill up to and including ARG'th occurrence of CHAR.
Goes backward if ARG is negative; error if CHAR not found."
  (interactive "*p\ncZap to char: ")
  (kill-region (point)
               (progn
                 (search-forward
                  (char-to-string char) nil nil arg)
                 (point))))
```

8.1.1 interactive 表达式

zap-to-char 函数中的交互(interactive)表达式如下所示：

```
(interactive "*p\ncZap to char: ")
```

括号中的部分 “*p\ncZap to char:”，指定了三件事情。第一，也是最简单的，即星号“*”，它意味着，如果缓冲区是只读的，就产生一个错误信号。这就是说，如果你试图在一个只读缓冲区中使用 zap-to-char 函数，你将无法剪切任何文本，并且你将收到一个这样的消息：

⊖ 这里是指英文原书中此处的句子：“Thus, if the cursor were at the beginning of this sentence...”。——译者注

"Buffer is read-only"，你的终端还可能会对着你鸣叫报警。

"*p\ncZap to char:"的第二部分是字符"p"。这部分以一个换行符"\n"结束。小写"p"是指传送给函数的第一个参量将是一个处理过的前缀参量的值。前缀参量用 C-u 以及其后的一个数来传送，或者用 M- 和一个数来传送。如果不带前缀参量交互地调用这个函数，默认值 1 将被传送给这个函数。

"*p\ncZap to char:"的第三部分是"cZap to char:"。在这一部分中，小写"c"是指交互表达式希望产生一个提示并且后续的参量将是一个字符。提示信息是跟在"c"之后的字符串"Zap to char:"。(冒号后面带一个空格会使提示信息更好看。)

所有这些，都是为 zap-to-char 函数准备参量。至此，这些参量都有了正确的类型，并显示给用户一个提示信息。

8.1.2 zap-to-char 函数体

zap-to-char 函数体包含了从光标的当前位置(即位点)直到(并包含)指定字符这一区域剪切文本的代码。代码的第一个部分如下所示：

```
(kill-region (point) ...
```

(point) 就是光标所处的当前位置，即位点。

代码的下一个部分是一个使用 progn的表达式。progn 表达式的主体由 search-forward 和 point 函数组成。

学习了 search-forward 函数之后，就容易理解 progn 是如何工作的了。因此我们将先学习 search-forward函数，然后再学习 progn 函数。

8.1.3 search-forward 函数

search-forward 函数是用于定位 zap-to-char 函数中被截取的字符的。如果查询成功，search-forward 函数就在目标字符串中最后一个字符处设置位点(在这个例子中，目标字符串只有一个字符)。如果查询是朝后进行的，search-forward 函数就在目标字符串的第一个字符处设量位点。同样，search-forward 函数返回 t值表示查询成功。(移动位点只是这个函数的附带效果。)

在 zap-to-char 函数中，search-forward 函数如下所示：

```
(search-forward (char-to-string char) nil nil arg)
```

search-forward函数有 4 个参量：

1)第一个参量是目标，就是所要查找的内容。这个参量必须是一个字符串，如"z"。

执行时，传送给 zap-to-char 函数的参量是一个单字符。由于计算机本身工作原理的限制，Lisp 解释器认为单个字符与一个字符串是不同的。在计算机内部，单个字符与一个仅仅包含单个字符的字符串有不同的存储格式(单个字符能用一个字节精确地记录，但是一个字符串可能很长也可能很短，计算机需要为此做准备)。因为 search-forward 函数是查询一个字符串的，所以zap-to-char 函数接收的、作为其参量的字符，必须在计算机内从一种格式转

换成另外一种格式，否则 search-forward 函数将会无法工作。char-to-string 函数就是用于完成这种转换工作。

2) 第二个参量绑定查询范围；它被指定为缓冲区中的某个位置。在这个例子中，查询能到达缓冲区末尾，因此没有设置任何绑定，第二个参量就是空（nil）。

3) 第三个参量告诉这个函数如果查询失败应该怎么办——可以发出一个出错信号(并打印一条消息)，也可以返回空值 (nil)。如果第三个参量被设置成空 (nil)，就是告诉这个函数如果查询失败就发出一个出错信号。

4) 第四个参量是重复计数值——待查找字符串出现的次数的计数。这个参量是可选的，如果在调用这个函数时没有给定计数值，就使用默认值 1。如果这个参量是一个负数，查询就朝后进行。

用一个模板形式来分析的话，search-forward 函数就是这样：

```
(search-forward "target-string"
                limit-of-search
                what-to-do-if-search-fails
                repeat-count)
```

下面我们将学习 progn 函数。

8.1.4 progn 函数

progn 函数使其每一个参量被逐一求值并返回最后一个参量的值。前面若干表达式的求值，仅仅是作为函数的附带效果，这些值被统统扔掉了。

progn 表达式的模板很简单：

```
(progn
  body...)
```

在 zap-to-char 函数中，progn表达式要完成两件事情：在正确的位置设置位点，将位点返回以使 kill-region 函数知道要剪切到什么地方。

progn 表达式的第一个参量就是 search-forward。当 search-forward函数在文本中找到目标字符串时，它就在文本中的目标字符串最后一个字符处设置位点（在这个例子中目标字符串只有单个字符）。如果查询是朝后进行的，search-forward 函数就在目标字符串的第一个字符处设置位点。位点的设置和变动也只是这个函数的附带效果。

progn 表达式的第二个也是最后一个参量是表达式 (point)。这个表达式返回位点的值，在这种情况下就是由 search-forward 函数移动过的位点的值。这个值由 progn 表达式返回，并传递给 kill-region 函数作为其第二个参量。

8.1.5 总结zap-to-char函数

现在已经了解了 search-forward 函数和 progn 函数是如何工作的，因此也就能分析 zap-to-char 函数是如何工作的了。

当 zap-to-char 函数被调用时，给 kill-region 的第一个参量是光标所在的位置，也

就是当时位点的值。在 progn 表达式内部，search-forward函数将位点移动到查找到的字符后，(point) 表达式则返回位点的值。kill-region函数将这两个值结合起来，第一个值作为要剪切部分的开始，第二个值作为要剪切部分的末尾，然后将这个区域内的文本剪切掉。

progn 函数是需要的，因为 kill-region 命令需要两个参量。如果 search-forward 函数和 (point) 表达式作为它另外两个参量，它将运行失败。progn 表达式是 kill-region 命令的一个参量而不是两个，它的返回值正是kill-region 命令需要的第二个参量。

8.1.6 第18版中 zap-to-char 函数的实现方法

在第18版中，zap-to-char 函数的实现方法与 第19版中这个函数的实现方法稍有不同：它剪切的文本不包含指定字符；并且当没找有到指定字符时就剪切到缓冲区末尾。

产生达种不同的原因在于 kill-region 命令的第二个参量。这个参量在第19版中是这样的：

```
(progn
  (search-forward (char-to-string char) nil nil arg)
  (point))
```

而在第18版中，则是下面这个样子的：

```
(if (search-forward (char-to-string char) nil t arg)
    (progn (goto-char
            (if (> arg 0) (1- (point)) (1+ (point))))
           (point))
  (if (> arg 0)
      (point-max)
    (point-min)))
```

这部分代码看起来相当复杂，但是如果将其一部分一部分分解开来分析，就容易理解了。

代码的第一部分是：

```
(if (search-forward (char-to-string char) nil t arg)
```

用 if 表达式模板来分析，就是：

```
(if able-to-locate-zapped-for-character-and-move-point-to-it
    then-move-point-to-the-exact-spot-and-return-this-location
  else-move-to-end-of-buffer-and-return-that-location)
```

对这个 if 表达式的求值，就给出了 kill-region 函数的第二个参量。因为它的第一个参量是位点，因此这个过程使 kill-region 函数可以将位点和指定字符之间的文本全部剪切掉。

我们已经描述了 search-forward 函数如何将移动位点作为它的附带效果完成的这一过程。在这个函数中，如果查找成功，search-forward 函数的返回值就是 t。否则，根据 search-forward 函数的第三个参量的不同，它要么返回 nil 值，要么产生一个错误消息。在这个例子中，t 是它的第三个参量，这使得 search-forward 函数在查找失败时返回 nil 值。就值我们将要看到的，可以容易地编写代码来处理函数返回 nil 值的达种情况。

在第18版的 zap-to-char 函数的实现中，if表达式将查询表达式作为其真假测试表达式。如果查询成功，Emacs 就对 if 表达式的 then 部求值；另一方面，如果查询失败，Emacs 就对 if 表达式的 else 部求值。

在 if 表达式中，当查询成功时，progn 表达式被执行——也就是说，它就像一个程序一样被运行。

前面已经讲过，progn是一个函数，它使其中的参量逐一被求值，并返回最后一个参量的值。前面的其他表达式仅仅作为附带效果而被求值。它们产生的值被统统扔掉了。

在这个版本的 zap-to-char 中，当查询函数 search-forward 找到它要查询的字符时，progn 表达式就被执行。这个 progn 表达式要完成两件事情：在正确的位置设置位点，返回位点的值以使 kill-region 知道要剪切到何处。

之所以需要 progn 表达式中的代码，是因为当 search-forward 函数找到指定字符串时，它就在目标字符串的最后一个字符后设置位点（在这个例子中，目标字符串只有一个字符）。如果是朝后查询的，则在目标字符串的第一个字符处前设置位点。

然而，这个版本的 zap-to-char 函数并不剪切最后匹配的字符。例如，如果zap-to-char 函数要剪切直到“z”的所有文本，实际上它并不剪切 “z”字符。因此，位点要仅仅移动到匹配字符不被剪切的位置。

8.1.7 progn表达式主体

progn 表达式的主体包含两个表达式。若要展开来详细描绘两个版本中 progn 表达式的不同之处，并加上注释，这个版本的 progn 表达式如下所示：

```
(progn
  (goto-char                    ; First expression in progn.
      (if (> arg 0)             ; If arg is positive,
          (1- (point))          ;   move back one character;
        (1+ (point))))          ;   else move forward one character.

  (point))                      ; Second expression in progn:
                                ;   return position of point.
```

这个 progn 表达式是这样工作的：当查询是朝前进行的（arg是正值），Emacs就在查找到的字符后面设置位点。通过将位点向后移动一个位置，查找到的字符就不被剪切。在这个例子中，progn 表达式应变成这样：(goto-char (1- (point)))。这个表达式将位点后移一个字符（1-函数从其参量中减1，就像1+函数往其参量中加1一样）。另一方面，如果传递给 zap-to-char 函数的参量是负数，查询就是朝后进行的。if 表达式检查到这一点，因此表达式实际上就成了：(goto-char (1+ (point)))。（1+函数往其参量中加 1。）

progn 表达式的第二个也是最后一个参量是表达式 (point)。这个表达式返回由 progn 表达式的第一个参量决定的位点的值。然后，if 表达式返回这个值。if 表达式是 kill-region 表达式的一部分，并将 if 表达式的这个返回值传递给 kill-region 表达式作为它

的第二个参量。

简要地说，这个函数的工作方式就是：kill-region 的第一个参量是当zap-to-char命令执行时光标所在的位置——也就是那个时候位点的值。然后，如果查询成功，查询函数将位点移动。progn 表达式将位点移动到匹配字符串刚好不被剪切的位置，并返回这时位点的值。最后，kill-region 函数剪切这段区域内的文本。

最后，if 表达式中的else部处理目标字符串没有被查到的情况。如果zap-to-char 函数的参量是正的（或者没有给出）而且查询失败，则当前位点到缓冲区可见区域末尾的所有文本都将被剪切。（如果没有设置变窄开启，就是从当前站点到整个缓冲区末尾的所有文本都将被剪切。）如果 arg 是负的，而又没有查找到目录字符串，则从当前位点到缓冲区可见区域开始处的文本都将被剪切。完成这些工作的代码是一个简单的 if 表达式：

```
(if (> arg 0) (point-max) (point-min))
```

这就是说，如果 arg 是一个正数，返回 point-max 的值；否则，返回 point-min 的值。

回顾起来，下面是包含 kill-region 表达式的代码(带有注释)：

```
(kill-region
 (point)                        ; beginning-of-region
 (if (search-forward
      (char-to-string char)     ; target
      nil                       ; limit-of-search: none
      t                         ; Return nil if fail.
      arg)                      ; repeat-count.
     (progn                     ; then-part
       (goto-char
        (if (> arg 0)
            (1- (point))
          (1+ (point))))
       (point))

   (if (> arg 0)                ; else-part
       (point-max)
     (point-min))))
```

通过比较你可以看到：第19版中 zap-to-char 函数的实现代码比第18版中该函数的实现代码少一些，但是更简洁。

8.2 kill-region函数

zap-to-char 函数使用了 kill-region 函数。这个kill-region函数很简单，就是删去文档字符串的一部分。其代码如下：

```
(defun kill-region (beg end)
  "Kill between point and mark.
The text is deleted but saved in the kill ring."
```

```
(interactive "*r")
(copy-region-as-kill beg end)
(delete-region beg end))
```

一个要特别注意的地方是，这个函数使用了 delete-region和 copy-region-as-kill 函数。这些函数将在接下来的章节描述。

8.3 delete-region函数：接数C

zap-to-char命令使用了 kill-region 函数，而 kill-region 函数又使用了两个其他的函数：copy-region-as-kill 和 delete-region。copy-region-as-kill函数将在随后小节中描述，它的作用是将某个区域中的文本复制一份到kill环中，因此这份文本可以重新找回来。(参见8.5节“copy-region-as-kill函数”。)

delete-region 函数删除一个区域中的内容，而且你无法找回它。

不像在这里讨论的其他函数，delete-region 函数不是用 Emacs Lisp 编写的，它是用 C 语言编写的，并且是 GNU Emacs 系统的一个基本函数。因为它非常简单，所以我就从 Lisp 中岔开来讲讲这个 C 语言函数。

就像许多其他的 Emacs 基本函数一样，delete-region是作为一个 C 语言宏的实例来被编写的，一个宏就是一个代码模板。这个宏的第一个部分如下所示：

```
DEFUN ("delete-region", Fdelete_region, Sdelete_region, 2, 2, "r",
  "Delete the text between point and mark.\n\
When called from a program, expects two arguments,\n\
character numbers specifying the stretch to be deleted.")
```

在没有深入到这个宏编写过程的细节之前，首先要指出的是这个宏是以 DEFUN 开始的。之所以选择 DEFUN 这个词，是因为它完成 Lisp 中 defun 相同的事情。DEFUN 一词后面的括号内跟着七个部分：

- 第一个部分是 Lisp 中的函数名，在这个例子中就是 delete-region。
- 第二部分是 C 语言中的函数名，即 Fdelete_region。习惯上，它以“F”开头。因为 C 语言中函数名不使用连字符，所以使用下划线。
- 第三部分是 C 常数结构名，这些常数结构在函数内部记录信息。它是 C 语言中的函数名，但是它以字符“S”开头而不是以“F”开头。
- 第四和第五部分指定了函数中允许的参量数目的最小值和最大值。在这个例子中，这个函数需要两个参量。
- 第六部分就像 Lisp 的一个函数中跟在interactive 说明之后的参量那样：要么是一个字符，要么是一个提示信息。在这个例子中，字符是“r”，它是指函数的两个参量将是一个缓冲区中某个区域的开始和结束的位置。在这段代码中，没有提示信息。
- 第七部分是文档字符串。除了每一个换行符都必须显式地写成 “\n”的形式外，它与 Emacs Lisp 中编写的函数的文档就没有别的不同之处了。

随后就是正式的参数（每个参数都有对这个参数的类型进行说明的语句），然后就是这个宏的主体部分。对 delete-region 函数而言，这个宏的主体包含了如下三行：

```
validate_region (&b, &e);
del_range (XINT (b), XINT (e));
return Qnil;
```

其中的第一个函数 validate_region 检查传递来的值的类型，判断它们作为缓冲区中一个区域的开始和结束值是否正确，是否在正确的范围之内。第二个函数 del_range 实际上真正完成删除文本的功能。如果这个函数正确地删除了文本，则第三行中的函数返回 Qnil 来表示它已经顺利完成了删除任务。

del_range 函数是一个复杂的函数，在此不再继续深入研究。它的作用是更新缓冲区并完成一些其他的事情。然而，看看传递给它的两个参量还是值得的。这两个参量是 XINT(b) 和 XINT(e)。在 C 语言中，b 和 e 是两个 32 位的整数，它们记录要删除的区域的开始和结束的位置。但是就像 Emacs Lisp 中的其他数一样，32 位中只有 24 位是用于存放实际的数值，剩下的 8 位用于跟踪这些数的类型和其他信息。(在某些机器中，只有 6 位能用于这种目的。)在这个例子中，8 位用于指出这些数是指缓冲区中的位置。当一个数中的某些位用于这样的目的时，这就被称作一个标签 (*tag*)。在 32 位数据中使用 8 位标签使得这样编写 Emacs 代码的速度比用其他方式编写代码的速度更快。另一方面，由于实际数字只占用了 24 位，因此 Emacs 缓冲区近似地限制在8MB。(通过在编译前在“emacs/src/config.h”中定义 VALBITS 和 GCTYPEBITS，你就可以增加缓冲区的最大容量。参见 Emacs 发行版本中的“emacs/etc/FAQ”文件中的注释。)

“XINT”是一个 C 语言宏，它从 32 位的 Emacs 对象中提取 24 位，用于其他目的的8位就被扔掉了。因此，del_range(XINT(b)、XINT(e)) 删除以“b”开始以“e”结束的区域中的内容。

从开发 Lisp 的人员的角度来看，Emacs 是相当简单的；但是隐藏在其中的内容却非常复杂玄妙。

8.4 用 defvar 初始化变量

不像 delete_region 函数是用 C 语言编写的，copy-region-as-kill 函数是用 Emacs Lisp 编写的。这个函数的功能就是拷贝缓冲区中的一个区域并将其保存到被称为 kill-ring 的变量中。这一节就描述这个变量如何被创建和如何被初始化。

(再一次提醒你，kill-ring这个术语确属于用词不当。从缓冲区中剪切出去的文本能够被找回来。它不是尸体之环，而是一个可以复活的文本环。)

在 Emacs Lisp 中，一个变量(如kill-ring)是通过使用 defvar 特殊表而被创建和赋初值的。这个特殊表的名字来源于“define variable”(定义变量之意)。

defvar 特殊表与给一个变量赋值的 setq 函数相似。它和 setq 有两个不同之处。第一，它只对无值的变量赋值。如果变量已经有一个值，defvar特殊表就不会覆盖已经存在的值。第二，defvar 特殊表有一个文档字符串。

可以用 describe-variable 函数查看任何一个变量的当前值，describe-variable 这个函数常常通过键入 C-h v 来激活。如果键入 C-h v，然后在提示下输入 kill-ring (并

回车），将看到当前的 kill 环中的内容——可能是相当多的。相反，如果在这个 Emacs 进程中除了阅读之外什么也没有做，你可能什么也看不到。在“*Help*”缓冲区的末尾，将看到 kill-ring 的文档：

```
Documentation:
List of killed text sequences.
```

kill环是用下面的方式由 defvar 定义的：

```
(defvar kill-ring nil
  "List of killed text sequences.")
```

在这个变量定义中，变量被初始化为 nil，这是有意义的，因为如果你什么也没有保存，当你用 yank 命令时就无需返回任何东西。文档字符串就像 defun中的文档字符串一样被编写。至于 defun 定义中的文档字符串，其第一行应当是一个完整的句子，因为有些命令(如 apropos)仅仅打印其中的第一行。后续的行不应缩排；否则当你使用 c-h v (describe-variable)时它们看起来很奇怪。

绝大多数变量是 Emacs 的内部变量，但是有一些是可以用 edit-options 命令方便地设置的。(这些设置仅仅在一个编辑过程中有效；要永久地设置一个值，可以编写一个“.emacs”文件。参见第16章，“配置你的‘.emacs’文件”。)

一个可以重新设置的变量，是用文档字符串的第一行前面加上星号“*”来使之与Emacs 中的其他变量区别开来。

例如：

```
(defvar line-number-mode nil
  "*Non-nil means display line number in mode line.")
```

这意味着你能够使用 edit-options 命令来改变 line-number-mode 变量的值。

当然，你也能够在一个 setq 表达式当中对line-number-mode 变量求值来改变这个变量的值。

```
(setq line-number-mode t)
```

参见1.9.2节，“使用setq函数”。

8.5 copy-region-as-kill函数

copy-region-as-kill函数拷贝缓冲区中的一个文本区域，并将其保存到kill-ring 变量中。

如果在调用 kill-region命令后马上调用 copy-region-as-kill函数，Emacs将会把这个新拷贝的文本追加到原来拷贝的文本后。这意味着如果要找回文本，你将得到所有的内容(包括原来拷贝的内容和新拷贝的内容)，另外，如果在调用copy-region-as-kill 函数之前有其他命令，这个函数就将文本拷贝到 kill 环的另外一个入口。

下面是第18版中的 copy-region-as-kill 函数的全部代码，在此增加了几个注释以使格式更清楚：

```
(defun copy-region-as-kill (beg end)
  "Save the region as if killed, but don't kill it."
  (interactive "r")

  (if (eq last-command 'kill-region)

      ;; then-part: Combine newly copied text
      ;;   with previously copied text.
      (kill-append (buffer-substring beg end) (< end beg))

    ;; else-part: Add newly copied text as a new element
    ;;   to the kill ring and shorten the kill ring if necessary.
    (setq kill-ring
          (cons (buffer-substring beg end) kill-ring))
    (if (> (length kill-ring) kill-ring-max)
        (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil)))
  (setq this-command 'kill-region)
  (setq kill-ring-yank-pointer kill-ring))
```

像通常一样，这个函数能被分成几个组成部分：

```
(defun copy-region-as-kill (argument-list)
  "documentation..."
  (interactive "r")
  body...)
```

从这里可以清楚地看到，函数的参量是 beg 和 end，并且函数是交互的，带有“r”参数。因此函数的两个参量必须指向一个区域的开始和结束位置。如果你是从头阅读这份文档，理解这几部分几乎就像是例行公事一样简单。

除非你记得“kill”一词与它的原意有一定的差别，否则函数的文档会使你有些糊涂。

函数体开始于一个 if 从句。这个从句所做的工作就是判别两种不同情况：这个命令是否是在前面一个 kill-region 函数后面立即执行的。如果是，则新拷贝的文本被追加到原来拷贝的文本后。否则，就作为一个与原来的文本分开的单独文本插入 kill 环的开头。

函数的最后两行是两个 setq 表达式。其中一个表达式将变量 this-command 设置为kill-region，另外一个表达式将变量 kill-ring-yank-pointer 指向 kill 环。

copy-region-as-kill 函数体将在下面详细讨论。

copy-region-as-kill 函数体

编写copy-region-as-kill 函数是为了使在一行中两次或者多次剪切的文本最终将重新组合到 kill 环的同一个入口。如果要从 kill 环找回文本，则会得到整个文本。而且，如果是从当前光标处朝前剪切文本，则剪切掉的文本将加到原来剪切文本的末尾处；如果是从当前光标处朝后剪切文本，则剪切掉的文本将加到原来剪切文本的开始处。这就是说，kill 环中的文字仍然是以正常的次序存放的。

copy-region-as-kill 函数使用了两个变量存放当前和之前的一个 Emacs 命令。这两个变量是 this-command 和 last-command。

正常情况下，每当一个函数执行时，Emacs 将 this-command变量设置为正被执行的函数（在这个例子中就是 copy-region-as-kill）。同时，Emacs 将变量 last-command 设置为变量 this-command 原来的值。然而，copy-region-as-kill 命令就不同，它将this-command 变量设置成 kill-region，这是调用copy-region-as-kill的函数名。

在 copy-region-as-kill 函数体的第一部分，if 表达式判定 last-command 变量的值是否是 kill-region。如果是，这个表达式的 then 部被求值，它使用kill-append 函数将拷贝的文本追加到 kill 环中第一个元素的文本之后。另一方面，如果变量 last-command 的值不是 kill-region，copy-region-as-kill 函数在kill环中加入一个新的元素。

这个if 表达式如下所示，它使用了一个我们未曾看到过的 eq 函数：

```
(if (eq last-command 'kill-region)
    ;; then-part
    (kill-append (buffer-substring beg end) (< end beg))
```

eq 函数测试其第一个参量是否与其第二个参量是同一个 Lisp 对象。eq 函数与equal 函数类似，它们在用于测试是否相等方面是一样的；但是，它们在判定不同名的两种表示所对应的 Lisp 对象是否是计算机中的同一个对象时是不同的。equal 函数判定两个表达式的结构和内容是否完全等同。

1. kill-append 函数

kill-append 函数如下所示：

```
(defun kill-append (string before-p)
  (setcar kill-ring
          (if before-p
              (concat string (car kill-ring))
              (concat (car kill-ring) string))))
```

可以一部分一部分地分析这个函数。其中，setcar 函数使用 concat 函数将新的文本追加到 kill环的 car 中（即第一个元素）。它是否追加或者前插文本依赖于 if 表达式的结果：

```
(if before-p                            ; if-part
    (concat string (car kill-ring))     ; then-part
  (concat (car kill-ring) string))      ; else-part
```

如果被剪切的文本是在最近一个命令剪切的文本之前，则这些文本应被插入到原来 kill 环中保存的内容之前。反过来，如果被剪切的文本是在最近被剪切的文本之后，则它应当被追加到原来那些文本之后。if 表达式依赖于 before-p 的判断来决定是否应将新保存的文本放在原来保存的文本之前还是之后。

符号 before-p 是 kill-append 函数的另外一个参量的名字。当 kill-append 函数被求值时，这个符号绑定到实际参量被求值后返回的值。在这个例子中，这就是表达式 (< end beg)。这个表达式并不直接决定剪切的文本是插入到前一个命令剪切的文本之前，还是追加到该文本之后。这个表达式所做的工作就是判定变量 end 的值是否小于变量 beg 的值。如果是，

则说明用户希望朝缓冲区开头剪切。同样地，如果对这个表达式求值的结果为“真”，则被剪切的文本将被插入到原来文本之前。另一方面，如果变量 end 的值大于变量 beg 的值，被剪切的文本就被追加到原来文本之后。

当新保存的文本要被插入到原有文本之前时，带新文本的字符串将被连接到老文本之前：

```
(concat string (car kill-ring))
```

但是，如果文本是被追加，则它将被连接到老文本之后：

```
(concat (car kill-ring) string))
```

为了理解这是如何实现的，首先需要回顾一下 concat 函数。concat 函数将两个文本字符串连接在一起，其结果是一个字符串。例如：

```
(concat "abc" "def")
     ⇒ "abcdef"

(concat "new "
        (car '("first element" "second element")))
     ⇒ "new first element"


(concat (car
        '("first element" "second element")) " modified")
     ⇒ "first element modified"
```

现在，我们能够来关注 kill-append 函数：它改变了 kill 环中的内容。kill 环是一个列表，其中的每一个元素都是保存的文本。setcar 函数实际上改变这个列表的第一个元素。它是通过使用 concat 函数将最新保存的文本连接到kill环的第一个元素来取代原来的第一个元素来实现的。最新保存的文本放在老文本之前或者之后，这依赖于在缓冲区中它是在原有文本之前还是之后。连接后的元素，成为 kill 环新的第一个元素。

顺便说一说，下面就是我的 kill 环开始处的内容：

```
("concatenating together" "saved text" "element" ...
```

2. copy-region-as-kill 函数中的 else 部

现在，回到 copy-region-as-kill 函数的解释：

如果最后一个命令不是 kill-region，则函数不是调用 kill-append，而是调用下面代码中的 else 部：

```
(if true-or-false-test
    what-is-done-if-test-returns-true
  ;; else-part
  (setq kill-ring
        (cons (buffer-substring beg end) kill-ring))
  (if (> (length kill-ring) kill-ring-max)
      (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil)))
```

else 部中的 setq 这一行，将在 kill 环上追加被剪切的字符串，并将这个新值赋给 kill 环。用一个小例子就可以看到它是如何运作的：

```
(setq example-list '("here is a clause" "another clause"))
```

键入 C-x C-e 对这个表达式求值后，可以对 example-list 求值并查看它所返回的值：

```
example-list
     ⇒ ("here is a clause" "another clause")
```

现在，通过对下面的表达式求值，就能够往这个列表中增加一个新的元素：

```
(setq example-list (cons "a third clause" example-list))
```

当对 example-list 求值时，将发现它的值是：

```
example-list
     ⇒ ("a third clause" "here is a clause" "another clause")
```

因而，通过 cons 函数，第三个元素增加到了 example-list 列表中。

除了用 buffer-substring 截取剪切区域中的文本之外，这个例子与在 copy-region-as-kill 函数中使用 setq 和 cons 函数的情况非常相似。这个语句重新写在下面：

```
(setq kill-ring (cons (buffer-substring beg end) kill-ring))
```

copy-region-as-kill 函数中 else 部的下一段是另外一个if 表达式。这个 if 表达式使 kill 环不致于过长。它是这样的：

```
(if (> (length kill-ring) kill-ring-max)
    (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil))
```

这部分代码检查 kill 环的长度是否大于最大允许的长度。最大允许长度就是kill-ring-max 的值（默认的情况下是30）。如果 kill 环的长度太长，这部分代码将 kill 环的最后一个元素设置为 nil。这是通过使用 setcdr 和nthcdr 函数来实现的。

我们首先来看看 setcdr 函数（参见7.5节“setcdr函数”）。这个函数设置一个列表的 cdr，就像 setcar 函数设置一个列表的 car 一样。然而在这个例子中，setcdr 并不是设置整个 kill 环列表的 cdr；nthcdr函数用于设置 kill 环的第二个元素到最后一个元素这个列表的 cdr——这意味着，如果第二个元素到最后一个元素的 cdr 是kill环的最后一个元素，它将设置 kill环的最后一个元素。

nthcdr 函数反复地取一个列表的 cdr——即它取一个列表的cdr的 cdr 的 cdr..... 这样重复 *N* 次，并返回最后的结果。

因此，如果现有一个4元素的列表，而假设只能有3个元素，则需要设置第二个元素到最后一个元素的 cdr为最后一个元素并赋 nil 值，从而缩短列表的长度。

依次对下面三个表达式求值就可以看到这一点。首先将变量 trees的值设置为(maple oak pine birch)，然后设置其第二次 cdr 的 cdr 为 nil，然后求 trees变量的值：

```
(setq trees '(maple oak pine birch))
     ⇒ (maple oak pine birch)
(setcdr (nthcdr 2 trees) nil)
     ⇒ nil

trees
     ⇒ (maple oak pine)
```

（由 setcdr 表达式返回的值是 nil，因为这就是 cdr 设置的值。）

再重复一下，在 copy-region-as-kill 函数中，nthcdr 函数重复取若干次的 cdr，其次数是kill环的最大允许长度减 1，并将那个元素的 cdr（其实这就是列表的最后一个元素）设置为 nil。这就可以避免 kill 环无限制地增长。

copy-region-as-kill 函数的倒数第二行是：

```
(setq this-command 'kill-region)
```

这一行代码既不属于内层 if 表达式，也不属于外层 if表达式，因此每当 copy-region-as-kill 函数被调用一次，这个表达式就被求值一次。在这里，我们发现此处this-command 变量被赋值为 kill-region。就像前面看到的，当执行下一个命令时，变量last-command将被赋为这个值。

最后，copy-region-as-kill 函数的最末一行是：

```
(setq kill-ring-yank-pointer kill-ring)
```

变量 kill-ring-yank-pointer 是一个全局变量，它被设置为 kill-ring。

虽然 kill-ring-yank-pointer 变量被称为“pointer”(指针)，但是它仅仅是一个像 kill 环这样的列表变量。然而，选择这样的名字是为了帮助人们理解如何使用这个变量。像 yank 和 yank-pop 这样的函数常使用这个变量（参见第10章，“找回文本”）。

yank 这些函数使我们将已经剪切的文本重新找回来。然而，在讨论 yank 命令之前，最好先学习列表是如何在计算机中实现的。这可以使像术语“指针”的使用这样神秘的内容变得清楚易懂。

8.6 回顾

下面是本章已经介绍过的一些函数的简单小结。

- cdr、car

car返回一个列表的第一个元素，cdr 则返回列表的第二个元素直到最后一个元素的列表。

例如，

```
(car '(1 2 3 4 5 6 7))
     ⇒ 1
(cdr '(1 2 3 4 5 6 7))
     ⇒ (2 3 4 5 6 7)
```

- cons

这个函数通过将它的第一个参量插入到它的第二个参量中来构造一个列表。

例如，

```
(cons 1 '(2 3 4))
     ⇒ (1 2 3 4)
```

- nthcdr

这个函数返回对一个列表求 *N*次 cdr 的值，也就是“剩余的剩余部分”。

例如，

```
(nthcdr 3 '(1 2 3 4 5 6 7))
     ⇒ (4 5 6 7)
```

• setcdr、setcar

setcar改变一个列表的第一个元素，而setcdr则改变一个列表的第二个到最后一个元素。

例如：

```
(setq triple '(1 2 3))

(setcar triple '37)

triple
     ⇒ (37 2 3)

(setcdr triple '("foo" "bar"))

triple
     ⇒ (37 "foo" "bar")
```

• progn

这个函数依次对其每一个参量求值，并返回最后一个参量的值。

例如：

```
(progn 1 2 3 4)
     ⇒ 4
```

• save-restriction

这个函数记录当前缓冲区中变窄开启是否设置，如果已经设置，就在对后续的参量求值之后恢复变窄开启。

• search-forward

这个函数查找一个字符串，并且如果找到这个字符串就移动位点。

这个函数有4个参量：

1) 要查找的字符串。

2) 查找的限制范围（可选）。

3) 如果查找失败应如何处理，是返回nil还是返回一个错误消息（可选）。

4) 重复查找多少次，如果这个参量的值是负的，就是朝后查找（可选）。

• kill-region、delete-region、copy-region-as-kill

kill-region函数将一个缓冲区中位点和标记之间的文本剪切掉，并将这些文本保存在kill环中，因此能够将它们重新找回来。

delete-region函数将缓冲区中位点和标记之间的文本移走并扔掉，不能够将它们再重新找回来。

copy-region-as-kill函数将缓冲区中位点和标记之间的文本拷贝到kill环中，从kill环中可以将它们重新找回来。这个函数不将缓冲区中的文本剪切掉。

8.7 查找练习

• 编写一个查找字符串的交互函数。如果找到需要的字符串，在其后设置位点并显示这样

一条消息："Found!"。（不要使用 search-forward 作为这个函数的函数名；如果使用了这样一个函数名，将覆盖 Emacs 的 search-forward 函数本身。可以使用如 test-search 这样的函数名。）

- 编写一个函数，这个函数在回显区打印 kill 环的第三个元素。如果 kill 环没有第三个元素，则打印一条适当的消息。
- 在第19.29版中，copy-region-as-kill 函数不再设置 this-command 变量。这种变化的后果是什么？要采取什么样的相应变化，才能达到同样的效果？