第12章 正则表达式查询

正则表达式查询在 GNU Emacs 中使用很广泛。forward-sentence 和 forward-paragraph 这两个函数充分展示了如何使用正则表达式进行查询。

在《*GNU Emacs*技术手册》中的“正则表达式查询”一节和《*GNU Emacs Lisp* 技术手册》中的“正则表达式”一节都有正则表达式查询的有关内容。在编写这一章时，假定读者对正则表达式已经有所了解。要记住的最重要的一点是，正则表达式允许按一定的模式查询，也允许完全按照字符串字面意义进行查询。例如，forward-sentence 函数查询能够标记句子结束的可能的字符模式，并将位点移动到句子末尾。

在展示 forward-sentence 函数代码之前，分析什么肯定是标记句子结束的模式是值得的。这个模式在下一节讨论，随后的一节是对这个正则表达式查询函数re-search-forward的描述。再后面一节描述 forward-sentence 函数。最后，在这一章的最后一节将讲解整个 forward-paragraph 函数。forward-paragraph-end函数是一个复杂的函数，其中引入了 Emacs Lisp 函数的几个新特征。

12.1 查询sentence-end的正则表达式

符号sentence-end被绑定到标记句子结束的模式上。这个正则表达式应是怎样的呢?

很清楚，一个句子可以以一个句点、一个问号或者一个感叹号结束。确实，只有当一个从句以上面三个符号之一结束才应被认为是一个句子的结束。这意味着，这个模式应当包括下面这个字符集：

```
[.?!]
```

然而，我们不希望 forward-sentence 函数仅仅移动到句点、问号或者感叹号之后，因为这样的字符可能出现在句子当中。例如，句点被用于缩写之后。因此，这种查询还需要其他信息。

根据惯例，在每个句子之后空上两个空格，在句子当中的句点、问号和感叹号之后仅空一个空格。因此，句点、问号和感叹号后有两个空格的话，这就是标记一个句子结束的一个好的指示器。然而，在一个文件中，两个空格可能用 TAB 键或者换行符代替了。这意味着，这个正则表达式应当包含这三种情况。这组可选方案就是：

```
\\($\\| \\|  \\)
       ^  ^^
       TAB  SPC
```

其中，“$” 符号表示一行的结束。在上面的例子中已经指出 TAB 键和两个空格插入到表达式中的情形。这两种情况都是将实际的字符插入到表达式中。

括号和垂直线之前的两个反斜线“\\”是需要的：在 Emacs 中，第一个反斜线引用第二个反斜线；第二个反斜线指出后续的字符（即括号或垂直线）是特殊字符。

而且，一个句子可以后接一个或者多个回车符，就像这样：

```
[
]*
```

就像制表符（TAB键）和空格一样，回车符(RET键)也是直接插入到正则表达式中的。上面这个星号表示回车符重复了零次或者多次。

但是一个句子结束并不仅仅是由句点、问号、感叹号及其后接的适当的空格组成的：空格之前可能还有一个右引号或者一个大右括号。确实，在空格之前可以有多于一个这样的引号或者大括号。这种情况需要下面的表达式：

```
[]\"')}]*
```

在这个表达式中，第一个“]”是这个表达式的第一个字符。第二个字符是“"”，它之前有一个“\”字符以告诉 Emacs 这个“"”字符不是特殊字符。最后三个字符是“'”、“)”和“}”。

所有这些表示与一个句子结束相匹配的正则表达式应当是什么样的。确实，如果对 sentence-end 函数求值的话，将发现它返回下面这样的值：

```
sentence-end
     ⇒ "[.?!][]\"')}]*\\($\\|      \\|  \\)[
]*"
```

12.2 re-search-forward 函数

re-search-forward 函数与 search-forward 函数非常相似。（参见8.1.3节，“search-forward 函数”）。

re-search-forward 函数查询一个正则表达式。如果查询成功，它就紧接在最后的目标字符后面设置位点。如果查询是朝后进行的，就在第一个目标字符之前设置位点。可以告诉 re-search-forward 函数在查询成功时返回“真”(t)。（移动位点只是这个函数的一个附带效果。）

就像 search-forward 函数，这个 re-search-forward 函数也有四个参量：

1) 第一个参量是函数要查询的正则表达式。这个正则表达式是一对引号之间的字符串。

2) 第二个参量是可选的，这个参量限制函数查询的范围；它是在缓冲区中指定一个位置的约束值。

3) 第三个参量是可选的，它告诉函数如何对查询失败作出响应：如果第三个参量是nil，函数在查询失败时产生一个错误消息；如果第三个参量是其他值，函数在查询失败时返回 nil，查询成功时返回t。

4) 第四个参量也是可选的，它是一个重复计数。一个负的重复计数使 re-search-forward 函数向后查询。

re-search-forward 函数的模板是：

```
(re-search-forward "regular-expression"
                   limit-of-search
                   what-to-do-if-search-fails
                   repeat-count)
```

第二、第三和第四个参量是可选的。然而，如果要给最后两个参量或者最后两个参量的任意一个参量传递一个值，必须要给其他前面的参量传递相应的值。否则，Lisp 解释器将不知道要将值传递给哪一个参量。

在 forward-sentence 函数中，正则表达式将是变量 sentence-end 的值，也就是：

```
"[.?!][]\"')}]*\\($\\|   \\|  \\)[
]*"
```

查询的范围至段落的末尾（因为一个句子不会超过一段）。如果查询失败，这个函数将返回 nil，重复计数将由函数 forward-sentence 的参量提供。

12.3 forward-sentence 函数

将光标移动到一个句子之前的命令，是展示如何在 Emacs Lisp 中使用正则表达式查询的简单例子。确实，这个函数看起来比它实际的更长、更复杂，这是因为这个函数设计成既可以朝前也可以朝后移动，作为可选项，也可以移过多个句子。这个函数通常绑定到命令键 M-e 上。

下面是 forward-sentence 函数的代码：

```
(defun forward-sentence (&optional arg)
  "Move forward to next sentence-end.  With argument, repeat.
With negative argument, move backward repeatedly to sentence-beginning.
Sentence ends are identified by the value of sentence-end
treated as a regular expression.  Also, every paragraph boundary
terminates sentences as well."
  (interactive "p")
  (or arg (setq arg 1))
  (while (< arg 0)
    (let ((par-bag
           (save-excursion (start-of-paragraph-text) (point))))
      (if (re-search-backward
           (concat sentence-end "[^ \t\n]") par-beg t)
          (goto-char (1- (match-end 0)))
        (goto-char par-beg)))
    (setq arg (1+ arg)))
  (while (> arg 0)
    (let ((par-end
           (save-excursion (end-of-paragraph-text) (point))))
      (if (re-search-forward sentence-end par-end t)
          (skip-chars-backward " \t\n")
        (goto-char par-end)))
    (setq arg (1- arg))))
```

这个函数初看起来很长，因此最好先来看看这个函数的骨架结构，然后再分析其“肌肉”——具体的内容。这个函数的骨架结构由最靠左边的几个表达式组成：

```
(defun forward-sentence (&optional arg)
  "documentation..."
  (interactive "p")
```

```
(or arg (setq arg 1))
(while (< arg 0)
  body-of-while-loop
(while (> arg 0)
  body-of-while-loop
```

这看起来简单多了。这个函数定义包含了文档、一个 interactive表达式、一个 or 表达式和两个 while 循环。

让我们依次来分析这几个部分。

文档部分是详尽的，也很容易理解。

这个函数有一个交互的声明：interactive "p"。这意味着，如果有前缀参量，就将其作为这个函数的参量传递给函数（这个参量将是一个数）。如果没有传递参量（参量是可选的）给这个函数，参量 arg 就被绑定到数值 1。当函数forward-sentence 是无参量、非交互调用时，arg 绑定到 nil。

or 表达式处理前缀参量。它所做的，要么是保持 arg的原值（仅当arg被绑定到一个值时），要么将arg的值设定为 1（在 arg 被绑定到 nil 的情况下）。

1. while 循环

在 or 表达式之后跟着两个 while 循环。第一个 while 循环有一个真假测试表达式，当传递给 forward-sentence 函数的前缀参量是一个负数时，这个测试表达式结果为“真”。这是为朝后查询设置的。这个循环的主体与第二个while从句的循环体很相似，但是并不完全一样。我们将跳过这个 while 循环而首先关注第二个 while 循环。

第二个 while 循环完成将位点前移的工作。它的骨架结构是这样的：

```
(while (> arg 0)                ; true-or-false-test
  (let varlist
    (if (true-or-false-test)
        then-part
      else-part
  (setq arg (1- arg))))         ; while loop decrementer
```

这个 while 循环是使用减量计数器的那一种循环（参见11.1.4节，“使用减量计数器的循环”。）它有一个真假测试表达式，只要计数器（在这里是变量arg）的值大于零，测试的结果就为“真”。它同时有一个递减器，在每一次循环中将计数器的值减 1。

如果没有前缀参量被传递给 forward-sentence 函数(这也是这个函数最常使用的方式)，这个 while 循环只运行一次，因为变量 arg 的值在这种情况下默认为 1。

while 循环体包含一个 let 表达式，这个 let 表达式创建并绑定一个局部变量。let 表达式中又有一个作为 let 表这式主体的 if 表达式。

这个 while 循环体如下所示：

```
(let ((par-end
       (save-excursion (end-of-paragraph-text) (point))))
  (if (re-search-forward sentence-end par-end t)
      (skip-chars-backward " \t\n")
```

```
    (goto-char par-end)))
```

其中，let表达式创建并绑定局部变量 par-end。就像将要看到的，这个局部变量是为了给正则表达式查询提供一个边界和限制而设计的。如果查询没能在段落结束时找到正确的句子，它将在段落末尾停止查询工作。

但是首先，要检查一下变量 par-end 是如何绑定到段落结束处的。这是let表达式在 Lisp 解释器完成对下面的表达式求值之后将其返回值赋给变量 par-end来完成的。

```
(save-excursion (end-of-paragraph-text) (point))
```

在这个表达式中，(end-of-paragraph-text) 将位点移动到段落末尾，(point) 返回位点的值，然后 save-excursion 恢复位点当初的值。因而，let表达式将 save-excursion的返回值（也就是段落结束的位置）赋给变量par-end。((end-of-paragraph-text) 函数使用了forward-paragraph函数，在后面将对这个函数作简短的介绍。)

Emacs 下一步计算 let 表达式主体，也就是下面的 if 表达式：

```
(if (re-search-forward sentence-end par-end t) ; if-part
    (skip-chars-backward " \t\n")              ; then-part
  (goto-char par-end)))                        ; else-part
```

这个 if 表达式测试它的第一个参量是否为“真”。如果为“真”，就对它的 then 部求值，否则 Emacs Lisp 解释器就对 else 部求值。if 表达式中的真假测试是一个正则表达式查询。

看起来也许奇怪，这个正则表达式查询像forward-sentedce函数的“实际工作”，但是这是这种操作在 Lisp 中实现的常用方法。

2. 正则表达式查询

re-search-forward 函数查询句子的结束，也就是查询由正则表达式sentence-end定义的模式。如果找到了这个模式——找到了句子的结束标志——re-search-forward 函数将完成两件事情：

1) re-search-forward 函数完成一个附带效果，将位点移动到当前找到的句子结束处。

2) re-search-forward 函数返回一个“真”值。这是一个由 if 函数接收的值，它意味着查询成功。

这个函数的附带效果——移动位点——是在 if 函数被递交由查询的成功结束所返回的值之前被完成的。

当 if 函数从对 re-search-forward的成功调用中接收到返回的“真”值时，就对 then 部，也就是表达式(skip-chars-backward “\t\n”)求值。这个表达式朝后移动并忽略所有空格、制表符（tab键）以及回车符，直到找到一个印刷字符为止，并将位点设置在这个字符之后。因为位点已经移动到标记句子结束的正则表达式模式末尾，这个动作就是将位点紧紧置于句子的结束打印字符之后，它通常就是一个句点。

另一方面，如果 re-search-forward 函数没能找到表示句子结束的相应模式，则函数返回“假”。查询失败使 if 函数对它的第三个参量，也就是对表达式(goto-char par-end)求值，即将位点移动到段落的结尾。

正则表达式查询特别有用，由re-search-forward 说明的查询模式也随处可见。在re-search-forward 中，查询就是通过if表达式的真假测试完成的。你将经常看到这个正则表达式，或编写包括这个正则表达式模式的代码。

12.4 forward-paragraph: 函数的金矿

forward-paragraph 函数将位点朝前移动到段落末尾。它一般绑定到M-}上。这个函数使用了大量很重要的函数，包括 let*、match-beginning 和looking-at 函数。

forward-paragraph 函数的定义比 forward-sentence 的函数定义长得多，因为它是查询一个段落。段落的每一行都可能以一个填充前缀开始。

一个填充前缀由一个字符串组成，这个字符串在每一行开始处重复出现。例如，在 Lisp 代码中，对于一个成段的注释，习惯上用“；；；”开始。在文本模式中，4个空格组成另外一个通用的填充前缀，用于创建一个缩排段落。（详细情况参见《*GNU Emacs* 手册》中的“填充前缀”一节。）

填充前缀的存在，意味着除了能够找到从最左边一列开始的段落的结束处之外，forward-paragraph 函数还一定能够找到缓冲区中所有或许多行都是以填充前缀开始的段落的结束处。

而且，有时实际上要忽略已经存在的填充前缀，特别是当空白行分割不同段落时。这是一个额外的复杂性。

在此不将 forward-paragraph 函数的所有代码打印出来，只是打印其中的一些部分。如果读代码前未做任何准备，这个函数将使你望而生畏。

从骨架结构上看，这个函数是这样的：

```
(defun forward-paragraph (&optional arg)
  "documentation..."
  (interactive "p")
  (or arg (setq arg 1))
  (let*
      varlist
    (while (< arg 0)          ; backward-moving-code
      ...
      (setq arg (1+ arg)))
    (while (> arg 0)          ; forward-moving-code
      ...
      (setq arg (1- arg)))))
```

这个函数的前面部分是通常的样子：函数的参量列表由一个可选参量组成，后面跟着函数文档。

interactive 中小写的“p”意味着，如果有前缀参量，就将其传递给这个函数。这个前缀参量是一个数，是关于要移动多少段落位点的重复计数。后面一行中的 or 表达式处理没有前缀参量传递给函数的情况，这种情况可能发生在这个函数被其他代码调用而不是交互地调用之中。这种情况在前面已经讲到过（参见12.3节，“forward-sentence函数”），这里就不再重

复。好了，到现在为止，这个函数中熟悉的部分都已讲完了。下面介绍其中的一些新东西。

1. let* 表达式

forward-paragraph 函数的后续一行开始于一个 let* 表达式。这是一种与我们前面接触过的表达式不同的表达式。符号 let* 不是let!

除了 Emacs 将变量依次赋值之外，let* 特殊表与 let 相似。let* 表达式中，变量列表中后面的变量可以使用前面的变量已经由 Emacs 设置的值。

在这个函数的 let* 表达式中，Emacs绑定了两个变量：fill-prefix-regexp和paragraph-separate。其中变量 paragraph-separate 的值依赖于变量 fill-prefix-regexp的值。

让我们逐个来看一看。符号 fill-prefix-regexp被设置为对下面的列表求值所返回的值：

```
(and fill-prefix
     (not (equal fill-prefix ""))
     (not paragraph-ignore-fill-prefix)
     (regexp-quote fill-prefix))
```

这个表达式的第一个元素是 and 函数。

and 函数不停地对它的参量求值，直到遇到一个返回值为 nil的参量为止。这时and 表达式的返回值就是 nil。然而，如果没有一个参量的值是 nil，则最后一个参量的值作为表达式的值被返回。(因为这个值不是 nil，它在 Lisp 中被认为是“真”。)换句话说，一个 and 表达式只有当它的所有参量的值都是“真”的时候，才返回“真”值。

在这个例子中，只有当下面四个表达式都产生一个“真”值(如非空值)时，变量 fill-prefix-regexp 才被绑定到一个非空值上；否则这个变量就被绑定到 nil。

- fill-prefix

当这个变量被求值时，填充前缀的值被返回。如果没有填充前缀，这个变量返回 nil。

- (not (equal fill-prefix “”))

这个表达式检查填充前缀是否是一个空白字符串（即没有字符的字符串）。空白字符串是没有什么用处的填充前缀。

- (not paragraph-ignore-fill-prefix)

当变量 paragraph-ignore-fill-prefix 已经被赋值为一个“真值”(如t)之后，这个表达式返回 nil。

- (regexp-quote fill-prefix)

这是 and 函数的最后一个参量。如果所有的参量都是“真”值，对这个表达式求值的结果将作为 and 表达式的返回值返回，这个返回值被绑定到变量 fill-prefix-regexp。

对 and 表达式的成功求值，就使变量 fill-prefix-regexp 绑定到 fill-prefix 的值上。这个值由 regexp-quote 函数修改。regexp-quote 函数所做的就是读入一个字符串并返回一个精确匹配这个字符串的正则表达式。这意味着，如果填充前缀存在，fill-prefix-regexp 将被设置为与填充前缀精确匹配的值。否则，这个变量被设置为 nil。

let* 表达式中的第二个局部变量是 paragraph-separete。它被绑定到对下面的表达式求值所返回的值上：

```
(if fill-prefix-regexp
    (concat paragraph-separate
            "\\|^" fill-prefix-regexp "[ \t]*$")
  paragraph-separate)))
```

这个表达式解释了为什么值用 let* 表达式而不是值用 let表达式。if 表达式的真假测试依赖于变量 fill-prefix-regexp 的值是 nil 还是其他值。

如果变量 fill-prefix-regexp 没有值(即它为nil)，Emacs 对 if 表达式的else部求值，并将变量paragraph-separate 绑定到它的局部值上。(paragraph-separate是一个正则表达式，是用于匹配分离的段落的正则表达式模式。)

但是如果 fill-prefix-regexp 变量确实有一个值(非空值)，Emacs 就计算if 表达式的 then 部，并将 paragraph-separate 绑定到一个正则表达式上，这个正则表达式包含fill-prefix-regexp作为模式的一部分。

特别地，paragraph-separate 被设置成匹配分离的段落的正则表达式的初始值，并在其后追加一个可供选择的表达式，这个追加的表达式由 fill-prefix-regexp 加上一个空行组成。其中的“^”符号表示 fill-prefix-regexp 必须是一行的开始，行末的可选空格由“[\t]*$”定义。“\\|”则定义了分离的段落的另外一种匹配方式。

现在进入 let* 表达式的主体。这个 let* 表达式主体的第一部分处理当这个函数被赋一个负参量并因此值位点朝后移动时的情况。我们的讨论将跳过这一部分。

2. 朝前查询的 while 循环

let* 表达式主体的第二部分处理位点的朝前移动。它是一个 while 循环。只要变量 arg 的值大于零，这个循环就不停地重复下去。在值用这个函数的绝大多数情况下，arg 变量的值是1，因此这个 while 循环体正好被求值 1 次，光标向前移动一个段落。

这部分代码处理三种情况：当位点处于两个段落之间时；当位点处于一个有填充前缀的段落当中时；当位点处于一个没有填充前缀的段落当中时。

while 循环看起来是这样的：

```
(while (> arg 0)
  (beginning-of-line)

  ;; between paragraphs
  (while (prog1 (and (not (eobp))
                     (looking-at paragraph-separate))
           (forward-line 1)))

  ;; within paragraphs, with a fill prefix
  (if fill-prefix-regexp
      ;; There is a fill prefix; it overrides paragraph-start.
      (while (and (not (eobp))
                  (not (looking-at paragraph-separate))
```

```
                    (looking-at fill-prefix-regexp))
            (forward-line 1))

        ;; within paragraphs, no fill prefix
        (if (re-search-forward paragraph-start nil t)
            (goto-char (match-beginning 0))
          (goto-char (point-max))))

      (setq arg (1- arg)))
```

我们能够立即看出，这是一个使用减量计数器的 while 循环，它使用表达式(setq arg (1- arg))作递减计数操作。循环体由三个表达式组成：

```
;; between paragraphs
(beginning-of-line)
(while
    body-of-while)

;; within paragraphs, with fill prefix
(if true-or-false-test
    then-part

;; within paragraphs, no fill prefix
  else-part
```

当 Emacs Lisp 解释器对 while循环体求值时，第一件事情就是计算表达式(beginning-of-line)的值，并将位点移动到这一行的开始。随后，函数中有一个内层的 while 循环。这个 while 循环是为将光标移出段落之间的空白处而设计的，如果光标碰巧正好在段落之间的空白处，就需要用到这个 while循环。最后，有一个 if 表达式，这个表达式完成真正将位点移动到段落末尾的操作。

3. 在段落之间的情况

首先，看看内层的 while 循环。这个循环处理位点处于段落之间的情况。它使用三个新函数：prog1、eobp 和 looking-at。

- prog1 函数与 progn 函数类似。不同之处在于 prog1 函数依次对它的参量求值并将其第一个参量的值作为整个表达式的值返回（progn 函数将它最后一个参量的值作为这个表达式的值返回）。prog1 的第二个和第三个参量只是作为附带效果被求值的。
- eobp 是“End Of Buffer P”（缓冲区的末尾）的缩写。当位点处于这个缓冲区末尾时，这个函数返回“真”。
- 当紧跟在位点之后的文本与传递给 looking-at 函数作为其参量的正则表达式匹配时，looking-at 函数返回“真”。

现在学习的这个 while 循环看起来是这样的：

```
(while (prog1 (and (not (eobp))
                   (looking-at paragraph-separate))
              (forward-line 1)))
```

这是一个没有循环体的 while 循环。循环的真假测试就是待计算的表达式：

```
(prog1 (and (not (eobp))
            (looking-at paragraph-separate))
       (forward-line 1)))
```

prog1函数的第一个参量是 and 表达式。在其内部有一个关于位点是否在缓冲区末尾的测试表达式，以及关于位点后的模式是否与分隔段落的正则表达式匹配的测试。

如果光标不在缓冲区的末尾，并且如果光标后面的字符表示两个分离的段落，则and 表达式为“真”。对 and 表达式求值之后，Lisp 解释器对 prog1 表达式的第二个参量 forward-line 求值。这个表达式使位点向前移动一行。然而，prog1 函数的返回值是其第一个参量的值，因此只要位点不在缓冲区的末尾而在两个段落之间时，while 循环将继续执行下去。最后，当位点被移动到一个段落时，and 表达式的值为“假”。然而要注意，forward-line 命令仍然要执行，这意味着，当位点从两个段落之间移动到一个段落当中时，它处于这个段落第二行的开头位置。

4. 在段落当中的情况

外层 while 循环的下一个表达式就是 if 表达式。当 fill-prefix-regexp 变量是一个非空值时，Lisp 解释器执行 if 表达式的 then 部。当fill-prefix-regexp 变量的值是 nil 时（也就是没有填充前缀时），Lisp 解释器执行 if 表达式的 else 部。

5. 没有填充前缀的情况

在没有填充前缀的情况下，代码是最简单的。这部分代码也由另一个内层 if 表达式组成，就像下面这样：

```
(if (re-search-forward paragraph-start nil t)
    (goto-char (match-beginning 0))
  (goto-char (point-max)))
```

这个表达式完成的工作实际上是 forward-paragraph 命令应当完成的主要工作：正则表达式查询，这个查询一直持续到下一段落的开始处。如果查询成功，则将位点移动到那里。但是，如果没有找到下一个段落的开始位置，它将位点移动到缓冲区中可访问区域的末尾。

这部分代码中你唯一不熟悉的部分就是 match-beginnning 的使用。这是另外一个新函数。match-beginning 函数最终返回一个数，这个数指定与最后一个正则表达式查询匹配的文本的开始处的位置。

因为这是一个具有代表性的查询过程，因此在这里使用了 match-beginning 函数。一个成功的朝前查询，不论它是一个普通的查询还是一个正则表达式查询，都将位点移动到找到的文本末尾。在这个例子中，一个成功的查询，会将位点移动到 paragraph-start 正则表达式模式末尾。这是下一段落的开始而不是当前段落的结束。

然而，我们的目的是要将位点置于当前段落的末尾，而不是将位点置子下一段落的开始。这两个位置是不同的，因为可能有几个空行在段落之间。

当给 match-beginning 函数传递参量 0 时，这个函数返回最近与正则表达式匹配的下一个段落的开始位置。在这个例子中，最近的正则表达式查询就是查询paragraph-start，因此 match-beginning 函数返回这个正则表达式模式的开始位置，而不是这个模式的末尾位置。开始的位置就是段落的末尾。

（顺便提一下，当给 match-beginning 传递一个正数作为参量时，这个函数将使位点置于

最后一个正则表达式中带括号的表达式处。这是一个很有用的函数。）

6. 有填充前缀的情况

刚才讨论的内层 if 表达式是外层 if 表达式的 else 部。这个 if 表达式判断是否存在填充前缀。如果有填充前缀，这个if表达式的 then 部被求值。就像下面这样：

```
(while (and (not (eobp))
            (not (looking-at paragraph-separate))
            (looking-at fill-prefix-regexp))
  (forward-line 1))
```

只要下面三种条件都为真，这个表达式就将位点向前一行一行地移动：

1) 位点不在缓冲区的末尾。

2) 位点后面的文本不分隔段落。

3) 位点之后的模式是有填充前缀的正则表达式。

除非你记得在 forward-paragraph 函数中位点被移动到一行的开始位置，否则最后这个条件可能使人疑惑。这意味着如果文本有填充前缀，looking-at 函数就将发现它。

7. 小结

总的来说，当朝前移动时，forward-paragraph 函数完成下面的工作：

- 将位点移动到一行的开始位置。
- 跳过段落之间的空行。
- 检查是否有填充前缀，如果有：

  ——只要不是分隔段落的空行，就要一行一行地向前移动。
- 但是，如果没有填充前缀：

  ——查询下一个段落开始的模式。

  ——移动到下一个段落开始模式处，这将是前一个段落的末尾。

  ——否则移动到缓冲区中可访问区域的末尾。

为复习方便，下面列出的是刚才讨论过的代码。为清楚起见，用缩进的方式将它们排列起来：

```
(interactive "p")
(or arg (setq arg 1))
(let* (
       (fill-prefix-regexp
        (and fill-prefix (not (equal fill-prefix ""))
             (not paragraph-ignore-fill-prefix)
             (regexp-quote fill-prefix)))

       (paragraph-separate
        (if fill-prefix-regexp
            (concat paragraph-separate
                    "\\|^"
                    fill-prefix-regexp
                    "[ \t]*$")
```

```
          paragraph-separate)))
  backward-moving-code (omitted) ...

  (while (> arg 0)                    ; forward-moving-code
    (beginning-of-line)

    (while (prog1 (and (not (eobp))
                       (looking-at paragraph-separate))
                  (forward-line 1)))
  (if fill-prefix-regexp
      (while (and (not (eobp))  ; then-part
                  (not (looking-at paragraph-separate))
                  (looking-at fill-prefix-regexp))
        (forward-line 1))
                                ; else-part: the inner-if
    (if (re-search-forward paragraph-start nil t)
        (goto-char (match-beginning 0))
      (goto-char (point-max))))

  (setq arg (1- arg)))))          ; decrementer
```

forward-paragraph 函数的完整定义，不仅包含这些朝前移动的代码，而且包含朝后移动的代码。

如果你在 GNU Emacs 中阅读这份文档，并且你想要看一看 forward-paragraph 函数的完整代码，可以键入 M-.(find-tag) 并在提示符下输入函数名来查看。如果find-tag函数首先问你“TAGS”表的文件名，就输入你的“emacs/src”目录中“TAGS”文件的文件名，这是一个类似“/usr/local/lib/emacs/19.23/src/TAGS”的路径名。(“emacs/src”目录的确切路径由你的Emacs拷贝安装的方式和位置决定。如果不知道路径，有时能用 C-h I 命令进入 Info，然后键入 C-x x-f 来查看“emacs/info”的实际路径。“TAGS”文件的路径对应着“emacs/src”的路径。然而，有时 Info 文件存放在其他地方。)

如果尚没有“TAGS”(标签文件)，你也能够创建自己的“TAGS”文件。

12.5 创建自己的“TAGS”文件

你能够创建自己的“TAGS”文件来帮助你访问源代码。例如，如果你有大量的文件在你的“~/emacs”目录中(就像我这样，我有 127个 .el 文件，而我希望载入17个)，你将发现如果创建了自己的“TAGS”文件，即使你使用 grep 命令或者别的命令来查找特定的函数，也能很容易地找到特定的函数。

你能够通过调用 etags 程序来创建自己的“TAGS”文件。这个程序是作为 Emacs 发行版本的一部分发行的。通常，etags 程序在 Emacs 安装时就会被编译和安装。(etags 不是 Emacs Lisp 的一个函数或者 Emacs 的一部分，它是一个 C 语言函数。)

要创建一个“TAGS”文件，首先要切换到需要创建这个“TAGS”文件的目录。在 Emacs 中，可以用 M-x cd 命令来切换到指定的目录，或者通过访问该目录下的一个文件，或者通过

用C-x d（dired）命令列出这个目录下的文件，来达到切换目录的目的。然后输入：

```
M-! etags *.el
```

来创建一个“TAGS”文件。etags 程序接收所有常用的 shell 的通配符。例如，如果需要为两个不同的目录创建一个共同的“TAGS”文件，输入以下这样一个命令就可以了，其中“.../elisp/”是第二个目录：

```
M-! etags  *.el ../elisp/*.el
```

输入

```
M-! etags --help
```

则可以列出 etags 程序能够接受的所有选项的列表。

这个 etags 程序能处理 Emacs Lisp、Common Lisp、Scheme、C、Fortran，Pascal、LaTeX以及大部分汇编语言。这个程序对不同的语言没有什么特殊的开关选项，它根据文件名以及文件的内容来识别文件所属的语言类型。

同样，当你自己编写函数代码并且希望参考自己已经编写好的函数时，etags程序就很有用。只要你在编写新的函数时不时地运行 etags 程序，就可以将这些新编写的函数变成“TAGS”文件的一部分。

12.6 回顾

以下是最近介绍的函数的简要总结。

• while

只要表达式主体的第一个元素的测试为“真”，这个表达式的主体就被不断地重复求值。最后返回 nil。（其中的表达式只是作为它的附带效果而被求值的。）

例如：

```
(let ((foo 2))
  (while (> foo 0)
    (insert (format "foo is %d.\n" foo))
    (setq foo (1- foo))))

     ⇒       foo is 2.
             foo is 1.
             nil
```

(insert 函数的工作就是在位点处插入它的参量。而 format 函数的作用就是以其参量的格式返回一个字符串，就像 message 函数一样，\n 产生一个新行。)

• re-search-forward

查询一种模式，并且如果找到这种模式，就将位点移动到那个位置。

同 search-forward 函数一样，这个函数也接受 四个参量：

1) 一个指定要查找的模式的正则表达式。

2) 可选的参量，即查询限制范围。

3) 可选参量，如果查询失败，返回 nil 值或者产生错误消息。

4) 可选参量，重复查询的次数；如果这个参量的值为负，表示查询朝后进行。

• let*

将局部变量绑定到指定的值上，然后对剩余的变量求值，它的返回值是最后一个参量的值。在绑定局部变量时，如果有的话就使用已经绑定的局部变量的值。

例如:

```
(let* ((foo 7)
       (bar (* 3 foo)))
  (message "'bar' is %d." bar))
    ⇒ 'bar' is 21.
```

• match-beginning

返回由最后一个正则表达式查询所找到的文本的开始位置。

• looking-at

如果位点后的文本与正则表达式匹配，就返回“真”(t)。

• eobp

如果位点是缓冲区的可访问区域的末尾，就返回“真”(t)。如果变窄没有开启，缓冲区中可访问区域的末尾就是缓冲区的末尾；如果变窄开启，它就是变窄部分的末尾。

• prog1

依次对其每一个参量求值，然后返回第一个参量的值。

例如:

```
(prog1 1 2 3 4)
     ⇒ 1
```

12.7 练习：使用 re-search-forward

- 编写一个函数，这个函数通过一个正则表达式来查询两个或者更多的连续空行。
- 编写一个函数，用来查询重复的单词，如“the-the”。关于如何编写一个正则表达式来匹配由两个相同部分组成的字符串，可以参见《*GNU Emacs* 手册》中的“正则表达式句法”一节。你能为此设计出几个正则表达式，一些比另一些更好。我使用的这个函数以及几个正则表达式附在附录A“the-the 重复单词函数”中。