第13章 计数：重复和正则表达式

重复和正则表达式查询是你在 Emacs Lisp 中编写代码时经常使用的功能强大的工具。这一章介绍在用 while 循环和递归方法构造的单词计数命令中正则表达式查询的使用情况。

在 Emacs 的标准发行版本中，包含了对一个区域内所有行进行计数的函数。然而，却没有对其中的单词进行计数的相应函数。

某些写作需要对单词进行计数。因而，如果你撰写一篇随笔，篇幅可以限制在800 字之内；如果你撰写一篇小说，可以规定自己一天撰写 1000字。Emacs 竟然缺少一个单词计数命令，这令我很奇怪。也许，人们使用 Emacs 的目的主要是编写代码或者那些不需要进行单词计数的文档，或者也许他们只是使用操作系统的单词计数命令。这个命令将文档中的字符数除以 5 得到单词数。无论如何，这里有一些单词计数命令。

13.1 count-words-region 函数

一个单词计数命令可以对一行、一个段落、一个区域或者一个缓冲区进行计数。这个命令应当包含哪些内容呢？你应当设计一个命令来对整个缓冲区中的单词进行计数。然而，Emacs 的传统鼓励程序富有弹性——即可以只对一节中的单词计数，而不是对一个缓冲区中的单词计数。因此，编写一个对一个区域计数的命令更有意义。一旦有一个 count-words-region 命令，如果愿意的话，也能够用 C-x h(mark-whole-buffer) 键序列对整个缓冲区中的单词计数。

很清楚，对单词计数是一个重复的动作：从这个区域的头部开始，对第一个单词开始计数，然后是第二个、第三个……等等，直到这个区域的末尾。这意味着单词计数使用递归的方法或者 while 循环是很理想的。

首先，将用 while 循环实现这个单词计数命令，然后用递归的方法实现这个命令。当然，这个命令将是交互的。

一个交互函数定义的模板总是这样的：

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

现在需要做的就是往其中填入适当的内容。

函数名应当直接表明自己的用途而无需另加说明，并且应当与已有的 count-lines-region 函数名相似。这可以使函数名易于记住。因此 count-words-region 是单词计数函数名的一个好选择。

这个函数对一个区域中的单词计数。这意味着参量列表一定要包含绑定到两个位置的符号，

这两个位置就是该区域的开始位置和结束位置。这两个位置可以分别被叫做“beginning”和“end”。文档的第一行应当是一个完整的句子，因为像 apropos 这样的命令将只打印函数说明文档的第一行。交互表达式的形式将是“(interactive "r")”，因为这将使 Emacs 将计数区域的开始位置和结束位置作为参量传递给这个函数。所有这些都是按部就班的。

需要编写函数体以使其完成这样三件任务：首先建立 while 循环的条件，其次是 while 循环，最后是发送一个消息给用户。

当用户调用 count-words-region 函数时，位点可能是在指定区域的开始，也可能是在末尾。然而，计数过程必须从区域的开始位置开始。这意味着，如果位点原来并不在指定区域的开始处，就要使位点移动到那里。执行（goto-char beginning）可以达到这个目的。当然，当函数执行完之后，要将位点返回到调用这个函数时的位置。为了这个原因，函数体必须包含在一个 save-excursion 表达式中。

函数体的中心部分由一个 while 循环组成，在这个循环中，一个表达式使位点一个单词一个单词地往前移动，另外一个表达式对移动的次数计数。只要位点还要继续往前移动，while循环的真假测试结果应当为“真”，当位点到达指定区域的末尾时，真假测试结果为“假”。

将使用 (forward-word 1) 作为表示向前一个单词一个单词地移动位点的表达式，但是如果使用一个正则表达式查询，就更容易看到 Emacs 是如何区分一个单词的。

正则表达式查询不仅查找一个模式，而且在找到之后将位点移动到其后。这意味着，一系列成功的正则表达式查询将使位点一个单词一个单词地向前移动。

实际的问题是，我们想要正则表达式查询直接跳过单词之间的空格和标点符号，就像跳过单词本身一样。一个拒绝跳过单词之间空格的正则表达式永远不会跳过多于一个的单词。这就是说，如果有空格和标点符号，正则表达式应当包含单词之后的这些空格和标点符号，把它们当做单词本身一样处理。（一个可能的情况是，一个单词可能结束一个缓冲区，而没有任何空格或者标点符号在其后，因此正则表达式的这个部分是可选的。）

因而，我们需要的这个正则表达式是一个模式，它定义一个或多个单词构词要素以及其后（可选的）的一个或多个不是单词构词要素的其他字符。这个正则表达式就是：

```
\w+\W*
```

缓冲区语法表决定了哪些字符是单词的构词要素而那些字符不是单词的构词要素。（有关语法的更多内容参见本书的14.2节，“单词或者符号是由什么构成的？”。同时，还可参见《*GNU Emacs手册*》的“语法表”一节以及《*GNU Emacs Lisp技术手册*》的“语法表”一节）。

查询表达式是这样：

```
(re-search-forward "\\w+\\W*")
```

（注意在“w”和“W”之前的成对的反斜线。对 Emacs Lisp 解释器来说，单个反斜线是有特殊意义的。它指后续的字符以不同于平常的方式被解释。例如，字符 “\n” 代表“newline”(即换行)，而不是一个反斜线和一个字母“n”。一行中两个连续的反斜线则代表一个通常的没有特殊意义的反斜线。）

需要一个计数器来对单词个数进行计数。这个变量必须首先设置为 0，然后Emacs 每循环一次，这个计数器就加 1。这个递增表达式很简单：

```
(setq count (1+ count))
```

最后，想要告诉用户在这个区域中有多少单词。message 函数就适合于用来向用户发出这种消息。这个消息必须是一个短语的形式，这样不管其中究竟有多少单词，它读起来总是正确的。我们不要这样的句子 "there are 1 words in the region"（有英文语法错误）。单数和复数之间的冲突是不符合英文语法规则的。用一个条件表达式根据这个区域中单词数目的不同给英文出不同的消息就能够解决这个问题。这里有三种可能：没有单词，一个单词，多于一个单词。这意味着在这里使用 cond 特殊表很合适。

所有这些综合起来就是下面这个函数定义：

```
;;; First version; has bugs!
(defun count-words-region (beginning end)
  "Print number of words in the region.
Words are defined as at least one word-constituent
character followed by at least one character that
is not a word-constituent.  The buffer's syntax
table determines which characters these are."
  (interactive "r")
  (message "Counting words in region ... ")

;;; 1. Set up appropriate conditions.
  (save-excursion
    (goto-char beginning)
    (let ((count 0))
;;; 2. Run the while loop.
      (while (< (point) end)
        (re-search-forward "\\w+\\W*")
        (setq count (1+ count)))

;;; 3. Send a message to the user.
      (cond ((zerop count)
             (message
              "The region does NOT have any words."))
            ((= 1 count)
             (message
              "The region has 1 word."))
            (t
             (message
              "The region has %d words." count))))))
```

就像写着的那样，这个函数可以正常工作，但是它并没有涵盖所有的情况。

count-words-region 中的空格bug

前一节描述的 count-words-region 函数命令中有两个bug，或一个bug的两种表现。第一，如果你标记的那个区域只有在某些文本当中包含空格，那个函数就会告诉你这个区域只包含一个单词。第二，如果你标记的区域只有在缓冲区的末尾或者在变窄的缓冲区的可以访问区域的末尾处包含一些空格，这个命令会显示一条这样的错误消息：

```
Search failed: "\\w+\\W*"
```

如果你是在 GNU Emacs 的 Info 中阅读这份文档，你可以自己测试这些 bug。

首先，以通常方式对上面那个函数定义求值以安装它。

如果愿意的话，你也可以通过对下面这个表达式求值来安装这个绑定键：

```
(global-set-key "\C-c=" 'count-words-region)
```

为了测试第一个 bug，在下面这一行的开始和末尾设置标记和位点，然后键入 C-c= (如果你没有绑定 C-c=，就使用 M-x count-words-region 命令)：

```
one   two  three
```

Emacs将正确地告诉你这个区域有三个单词。

重复这个测试，这次将标记置于这一行的开始，而将位点紧紧置于单词“one” 之前。再一次键入C-c= (或者 M-x count-words-region)。这时 Emacs 应当告诉你这个区域没有任何单词，因为它仅仅是由这一行的开始部分的空格组成的。但是，Emacs 会告诉你这里有一个单词！

再来进行第三次测试，将这一行拷贝到“*scratch*”缓冲区的末尾，然后在这一行的末尾输入几个空格。将标记置于单词“three”之后，并将位点置于这一行的末尾（也就是缓冲区的末尾）。再像前面那样键入 C-c= (或者 M-x count-words-region)，这时 Emacs 也应当告诉你这个区域没有任何单词，因为它只有这一行末尾的几个空格而已。但是，Emacs 显示一个错误消息：“Search failed”。

这两个bug源自同样一个问题。

首先，考虑这个 bug 的第一个表现。在这里，命令告诉你位于一行开始处的空格包含了一个单词。这是因为，M-x count-words-region 命令将位点移动到了区域的开始处。while 循环测试位点的这个值是否小于 end 的值，这个测试为“真”。相应地，正则表达式查找并找到第一个单词。Emacs 将位点置于这个单词之后，count 变量被设置为1。while 循环继续循环，但是这一次位点的值大于 end变量的值，退出循环。简单地说，正则表达式查询查找并找到一个被标记的区域之外的单词。

在bug的第二种表现中，指定的区域是缓冲区末尾的空格。Emacs 发出 “Search failed”的消息。这是因为 while 循环的真假测试值为“真”，因此查询表达式被求值，但是由于这个区域没有单词，因此查询失败。

在这个 bug 的两种表现当中，前一种是查询范围扩大了，后一种是试图到指定区域之外查询。

解决办法是限制查询的区域——这个动作相当简单，但是就像你将要看到的，它的实现方式实际上并非你想象的那么简单。

就像我们已经看到的，re-search-forward 函数接受一个查询模式作为它的第一个参量。但是除了这个强制性的参量之外，它还接受三个可选参量。可选的第二个参量绑定这个查询。可选的第三个参量，如果是 t，就使函数在查询失败时返回 nil 而不是发出一个错误消息。可选的第四个参量是一个重复计数器。（在 Emacs 中，你能够用 C-h f 加上函数名和回车符(RET键)得到一个函数的说明文档。）

在 count-word-region 函数定义中，指定区域的末尾的值是由变量 end 存放的，这个值作为一个参量传递给函数。因而，能够将 end 作为一个参量加入到正则表达式的查询表达式中：

```
(re-search-forward "\\w+\\W*" end)
```

然而，如果你仅在 count-words-region 的函数定义中做了这样的改变，并在一组空格上继续测试这个新版函数定义，你将得到一个错误消息声明查询失败："Search failed"。

出现这样问题的原因在于：查询被限制在这个区域中，而这其中没有组成单词的字符。因为查询失败，我们得到一个错误消息。但是我们不希望在这种情况下得到这样的一个错误消息。我们希望得到的消息是："The region does NOT have any words"。

这个问题的解决方法是为 re-search-forward 提供第三个参量 t，这使得函数当查询失败时返回 nil，而不是一个错误消息。

然而，如果你做出这样的改动并测试它，你将看到信息 " Counting words in region..."，并将一直看到这个消息，直到键入 C-g(keyboard-quit)为止。

这是因为，就像前面一样，查询被限制在指定区域，由于这个区域没有单词构词要素字符，因此查询就像希望的那样失败了。相应地，re-search-forward表达式返回 nil，什么也没有做。特别是，它没有移动位点，这原本是作为找到查询目标的一个附带效果实现的。在 re-search-forward表达式返回nil之后，while循环中的下一个表达式被求值，这个表达式使计数器递增1。然后循环继续下去。while循环中的真假测试结果一直为"真"，因为位点的值一直小于end的值（因为位点没有移动），循环一直进行下去。这就是你所看到的结果。

count-words-region函数需要另外一种改进，使 while 循环中的真假测试在查询失败时测试值为假。这时可以采用另外的方法，即要同时满足两种条件再对计数器做递增计算：位点必须还在指定区域中，并且查询表达式必须找到要计数的一个单词。

因为第一种和第二种条件必须同时为"真"，所以这两个表达式——区域测试表达式和查询表达式——能够用一个 and 函数结合成单独一个表达式，并嵌入到 while循环作为其真假测试表达式，就像这样：

```
(and (< (point) end) (re-search-forward "\\w+\\W*" end t))
```

(关于 and 函数的有关信息，参见12.4节"forward-paragraph:函数的金矿"。)

如果查询成功，re-search-forward 表达式返回t，作为其一个附带效果，位点也移动到相应的位置。因此，在找到单词的同时，位点从指定区域中一一移过。当查询表达式没能找到下一个单词，或者位点已经是指定区域的末尾时，真假测试结果为"假"，随即退出 while 循环，count-words-region 函数显示其消息。

做了这些最终的修改之后，count-words-region 函数就没有 bug 了（或者至少我没有发现它的 bug 了）。下面就是这个函数定义的完整代码：

```
;;; Final version: while
(defun count-words-region (beginning end)
  "Print number of words in the region."
  (interactive "r")
```

```
  (message "Counting words in region ... ")

;;; 1. Set up appropriate conditions.
  (save-excursion
    (let ((count 0))
      (goto-char beginning)

;;; 2. Run the while loop.
      (while (and (< (point) end)
                  (re-search-forward "\\w+\\W*" end t))
        (setq count (1+ count)))

;;; 3. Send a message to the user.
      (cond ((zerop count)
             (message
              "The region does NOT have any words."))
            ((= 1 count)
             (message
              "The region has 1 word."))
            (t
             (message
              "The region has %d words." count))))))
```

13.2 用递归的方法实现单词计数

与前面用 while 循环的函数类似，可以用递归的方法编写对单词计数的函数。下面，来看一看这是如何实现的。

首先，需要认识到 count-words-region 函数有三个任务：建立适当的计数条件，在指定的区域对单词计数，向用户发送消息告诉他们指定区域内有多少单词。

如果编写单独一个递归函数来完成所有的事情，那么将在每一次递归调用时都得到一个消息。如果这个区域有 13 个单词，就将得到 13 条消息，一个接着一个。实际上，我们不需要这样。相反，我们必须编写两个函数来完成这个工作，其中一个函数（递归函数）将在另外一个函数内部使用。一个函数将建立测试条件以及显示消息，另一个函数将返回单词计数。

让我们先从显示消息的函数入手。同时将继续称这个函数为 count-words-region。

这是用户将要调用的一个交互函数。确实，除了它将调用 recursive-count-words 函数来确定指定区域中有多少单词之外，它与这个函数的前面一个版本非常相似。

基于这个函数前面的版本，能够容易地构造出这个函数的模板：

```
;; Recursive version; uses regular expression search
(defun count-words-region (beginning end)
  "documentation..."
  (interactive-expression...)

;;; 1. Set up appropriate conditions.
  (explanatory message)
  (set-up functions...
```

```
;;; 2. Count the words.
    recursive call

;;; 3. Send a message to the user.
    message providing word count))
```

除了由递归调用返回的计数必须被传递给显示单词计数的消息这一点之外，这个定义看起来很直接。稍微思考一下就可以知道，这可以用 let 表达式实现：能够将 let 表达式的变量列表中的一个变量绑定到指定区域的单词数上，这个数值是由递归调用返回的；然后 cond 表达式就可以根据绑定的值向用户显示特定的消息。

人们常常认为 let 表达式中的这种绑定作用是这个函数的“第二位”的工作。但是，在这个例子中，你所认为的函数的“第一位”的工作——对单词计数，就是在 let 表达式中完成的。

使用 let 表达式，这个函数定义如下所示：

```
(defun count-words-region (beginning end)
  "Print number of words in the region."
  (interactive "r")

;;; 1. Set up appropriate conditions.
  (message "Counting words in region ... ")
  (save-excursion
    (goto-char beginning)

;;; 2. Count the words.
    (let ((count (recursive-count-words end)))

;;; 3. Send a message to the user.
      (cond ((zerop count)
             (message
              "The region does NOT have any words."))
            ((= 1 count)
             (message
              "The region has 1 word."))
            (t
             (message
              "The region has %d words." count))))))
```

下面，需要编写递归计数函数。

一个递归函数至少有三个部分：一个测试表达式、一个 next-step 表达式和递归调用。

测试表达式决定函数是否要再调用。因为是在一个指定区域中对单词计数，并且可以使用一个单词接一个单词地朝前移动位点的函数，所以测试表达式能够检查位点是否还在指定区域中。测试表达式应当找到位点的值并决定位点是在指定区域的末尾之前、正在末尾或者在末尾之后。能够用 point 函数来确定位点的值。很清楚，必须将指定区域的末尾的值作为一个参量传递给递归计数函数。

除此之外，测试表达式应当也能测试是否找到一个单词。如果没有找到单词，这个函数就不应当再调用它本身了。

next-step 表达式要改变一个值，以值当递归函数应当停止调用它本身时就可以停下来。更准确地说，next-step 表达式改变一个值以使之能在正确的时候使测试表达式让递归表达式停止调用函数本身。在这个例子中，next-step 表达式可以是一个单词接一个单词地朝前移动位点的表达式。

递归函数的第三部分就是递归调用。

在某处，我们也需要一个真正完成函数“工作”的那部分代码，也就是完成计数的代码。一个真正有用的部分！

但是，我们已经有了一个递归计数函数的骨架：

现在需要填满充实这个骨架。让我们首先从最简单的情况开始：如果位点在指定区域的末

```
(defun recursive-count-words (region-end)
  "documentation..."
   do-again-test
   next-step-expression
   recursive call)
```

尾或者超出了指定区域的末尾，就没有任何单词了。因此这时函数应当返回零。同样，如果查询失败，也没有任何单词计数，因此函数也应当返回零。

另一个方面，如果位点在指定的区域之内，查询也是成功的，函数应当继续调用本身。因而，测试表达式应当是：

```
(and (< (point) region-end)
     (re-search-forward "\\w+\\W*" region-end t))
```

注意，查询表达式是测试表达式的一部分——如果查询成功，函数返回t；如果查询失败，则返回 nil。（有关re-search-forward函数如何工作的解释，参见13.1节中的“count-words-region中的空格bug”小节。）

这个测试表达式是 if 函数的一个真假测试表达式。很清楚，如果测试表达式为“真”，if 从句的 then 部应当再一次调用函数；但是如果它为“假”，if 从句的 else 部应当返回零，因为不管位点是在指定区域之外或者查询失败，总之是没有别的单词了。

但是在考虑递归调用之前，需要先考虑 next-step 表达式。它是什么？有趣的是，它是测试表达式的查询部分。

除了为测试表达式返回 t或者 nil 之外，re-search-forward 函数在查询成功时还将位点向前移动，这是这个函数的一个附带效果。这个动作改变了位点的值，以使递归函数在位点完全移动到指定区域之外时，停止调用自身。因而，这个 re-search-forward 表达式就是 next-step 表达式。

从结构骨架上说，recursive-count-words 函数体如下所示：

```
(if do-again-test-and-next-step-combined
    ;; then
    recursive-call-returning-count
  ;; else
  return-zero)
```

那么它如何与计数的部分协作呢？

如果你不习惯编写递归函数，类似这样的问题可能会困扰你。但是它能够、也应当被系统

地解决。

我们知道，计数的方法应当通过某种途径与递归调用结合起来。确实，因为next-step 表达式将位点朝前移动一个单词，并且因为一个递归调用是对每一个单词都进行的，所以计数的方法必须是一个表达式，它将由recursive-count-words 调用返回的值增1。

考虑下面几种情况：

- 如果在指定区域中有两个单词，当函数对第一个单词计数时，函数应当返回的值是1 加上这个函数对区域中剩余的单词计数的结果（在这种情况下就是1）。
- 如果指定区域中只有一个单词，当函数对那个单词计数时，函数应当返回的值是 1 加上这个函数对区域中剩余的单词计数的结果（在这种情况下就是 0）。
- 如果指定区域中没有单词，则函数应当返回零。

从这个草案中，我们可以看到：在没有单词的情况下if 表达式的 else 部将返回零。这就是说，if 表达式的 then 部必须返回一个值，这个值是这个函数对剩余的单词计数的结果加1得到的。

这个表达式写在下面，其中 1+ 是一个对其参量加 1 的函数。

```
(1+ (recursive-count-words region-end))
```

而其中的 recursive-count-words 函数如下所示：

```
(defun recursive-count-words (region-end)
  "documentation..."

;;; 1. do-again-test
  (if (and (< (point) region-end)
           (re-search-forward "\\w+\\W*" region-end t))

;;; 2. then-part: the recursive call
      (1+ (recursive-count-words region-end))

;;; 3. else-part
    0))
```

让我们检查一下这个函数是如何工作的：

如果在指定的区域内没有单词，if 表达式的 else 部被求值，从而这个函数返回零。

如果指定区域中有一个单词，位点的值小于 region-end 变量的值，查询也是成功的。在这种情况下，if 表达式的测试结果为“真”，因此 if 表达式的 then 部被求值，也就是计数表达式被求值。这个表达式返回一个值（这也将是整个函数的返回值），这个值是 1 与递归函数调用返回的值之和。

同时，next-step 表达式已经使位点跳过第一个单词（并且在这种情况下也只有一个单词）。这意味着，当 (recursive-count-words region-end) 表达式第二次被求值时，作为递归调用的返回值，这时位点的值将等于或者大于region-end 变量的值。因此，这时 recursive-count-words 函数将返回零。零将被增大到1，并且前一个递归调用 recursive-count-words函数将返回 1加0，也就是 1，这就是正确的答案。

很明显，如果在指定区域中有两个单词，第一次调用recursive-count-words函数返回 1，加上对剩余区域作递归调用的 recursive-count-words 函数所返回的值——也就是 1加

1，即 2，这就是正确的答案。

类似地，如果在指定区域中有3个单词，recursive-count-words 函数的第一次调用返回 1 加上对剩余的两个单词的区域作递归调用的recursive-count-words 函数所返回的值——如此重复下去。

在此基础上加上完整的函数文档，这两个函数代码就完整地列于下面：

```
(defun recursive-count-words (region-end)
  "Number of words between point and REGION-END."
;;; 1. do-again-test
  (if (and (< (point) region-end)
           (re-search-forward "\\w+\\W*" region-end t))
;;; 2. then-part: the recursive call
      (1+ (recursive-count-words region-end))

;;; 3. else-part
    0))
```

调用这个递归函数的函数是：

```
;;; Recursive version
(defun count-words-region (beginning end)
  "Print number of words in the region.
Words are defined as at least one word-constituent
character followed by at least one character that is
not a word-constituent.  The buffer's syntax table
determines which characters these are."
  (interactive "r")
  (message "Counting words in region ... ")
  (save-excursion
    (goto-char beginning)
    (let ((count (recursive-count-words end)))
      (cond ((zerop count)
             (message
              "The region does NOT have any words."))
            ((= 1 count)
             (message "The region has 1 word."))
            (t
             (message
              "The region has %d words." count))))))
```

13.3 练习：统计标点符号的数量

用 while 循环编写一个函数，它对一个指定区域中的标点符号进行计数。标点符号包括句点、逗号、分号、冒号、感叹号、问号。另外，再用递归的方法编写一个这样的函数。