第17章 调 试

GNU Emacs 有两个调试器：debug 和 edebug。第一个调试器 debug 内置在Emacs之中，并且总是可以供你使用；第二个调试器 edebug 则是 Emacs 的一个扩充，这个调试器已经成为 Emacs 第19版的标准发行版本中的一个部分。

两个调试器都在《*GNU Emacs Lisp* 技术手册》中的“调试 Lisp 程序”一节中有详尽的描述。在这一章，将分别结合一个简单的例子简要地介绍这两个调试器。

17.1 debug

假设你已经编写了一个函数定义，这个函数将计算数字 1 到一个给定的数字之和，并返回这个结果。（这就是前面讲到的 triangle 函数。参见11.1.4节中“减量计数器的例子”小节的有关讨论。）

然而，你的函数定义有一个bug。你在输入“1-”的时候错误地输入了“1=”。下面是包含了这个错误的函数定义：

```
(defun triangle-bugged (number)
  "Return sum of numbers 1 through NUMBER inclusive."
  (let ((total 0))
    (while (> number 0)
      (setq total (+ total number))
      (setq number (1= number)))        ; Error here.
    total))
```

如果你在 Info 中阅读这份文档，可以用通常的方法对这个函数定义求值。你将看到 triangle-bugged 出现在回显区中。

现在用参量4传送给triangle-bugged函数，并对它求值：

```
(triangle-bugged 4)
```

你将得到下面的错误消息：

```
Symbol's function definition is void: 1=
```

实际上，对于这样的一个简单的 bug，传递这个错误消息的目的是告诉你改正这个函数定义所应当知道的内容。然而，假设你对此不确定，那怎么办呢？

你可以通过将 debug-on-error的值设置成 t 来开始调试：

```
(setq debug-on-error t)
```

这个表达式使 Emacs 在它下一次遇到一个错误的时候进入调试器。

通过将这个变量的值设置为 nil 就可以关闭它：

```
(setq debug-on-error nil)
```

将 debug-on-error 的值设置为 t，并对下面的表达式求值：

```
(triangle-bugged 4)
```

这一次，Emacs 将创建一个称为“*Backtrace*”缓冲区：

```
---------- Buffer: *Backtrace* ----------
Signalling: (void-function 1=)
  (1= number))
  (setq number (1= number)))
  (while (> number 0) (setq total (+ total number))
       (setq number (1= number))))
  (let ((total 0)) (while (> number 0) (setq total ...)
       (setq number ...)) total))
  triangle-bugged(4)
  eval((triangle-bugged 4))
  eval-last-sexp(nil)
* call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

(我已经将这个例子的格式稍微改动过，调试器不会将长的行折叠过来。)

你可以从这个缓冲区的底部开始往上读，它将告诉你Emacs 是如何一步一步地遇到错误。在这个例子中，Emacs 所做的就是交互地调用了 C-x C-e(eval-last-sexp)，这个命令导致对 triangle-bugged 表达式的求值。上面的每一行都告诉你后面 Lisp 解释器是如何求值的。

从缓冲区顶部开始的第三行是：

```
(setq number (1= number))
```

Emacs 试图对这个表达式求值。为了对这个表达式求值，它先对内部的表达式(1= number)求值，这个显示在从缓冲区顶部开始的第二行上。

```
(1= number)
```

这就是错误发生的地方，就像缓冲区最顶端的一行所示：

```
Signalling: (void-function 1=)
```

现在你能够改正这个错误，然后再对这个函数定义求值，并再一次运行你的测试表达式。

如果你正在 Info 中阅读这部分，现在就可以通过将其值设置为 nil来关闭 debug-on-error了：

```
(setq debug-on-error nil)
```

17.2 debug-on-entry

第二种启动 debug 的方法，是当你调用一个函数的时候进入调试器。调用debug-on-entry 就能够实现这一点。

键入：

```
M-x debug-on-entry RET triangle-bugged RET
```

现在，对下面的表达式求值：

```
(triangle-bugged 5)
```

Emacs 将创建名为“*Backtrace*”的缓冲区，并告诉你它将开始对triangle-bugged 函数求值：

```
---------- Buffer: *Backtrace* ----------
Entering:
* triangle-bugged(5)
  eval((triangle-bugged 5))
  eval-last-sexp(nil)
* call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

在“*Backtrace*”缓冲区中，键入 d。Emacs 将对 triangle-bugged 函数的第一个表达式求值，缓冲区内容将是：

```
---------- Buffer: *Backtrace* ----------
Beginning evaluation of function call form:
* (let ((total 0)) (while (> number 0) (setq total ...)
        (setq number ...)) total))
  triangle-bugged(5)
* eval((triangle-bugged 5))
  eval-last-sexp(nil)
* call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

现在，一次一次地慢慢键入 d，一共键入 8 次。每当你键入一次 d 键，Emacs 将对函数定义中的下一个表达式求值。最终，缓冲区内容将是：

```
---------- Buffer: *Backtrace* ----------
Beginning evaluation of function call form:
* (setq number (1= number)))
* (while (> number 0) (setq total (+ total number))
        (setq number (1= number))))
* (let ((total 0)) (while (> number 0)
        (setq total ...) (setq number ...)) total))
  triangle-bugged(5)
* eval((triangle-bugged 5))
  eval-last-sexp(nil)
* call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

最后，你再键入两次 d 之后，Emacs 将遇到错误，这时“*Backtrace*”缓冲区顶部的两行是：

```
---------- Buffer: *Backtrace* ----------
Signalling: (void-function 1=)
* (1= number))
...
---------- Buffer: *Backtrace* ----------
```

总之，通过键入 d 键，你可以一步一步地测试函数的每一个表达式。

通过键入 q 就可以退出“*Backtrace*”缓冲区，这个命令退出函数调试，但是并不取消 debug-on-entry。

要想取消 debug-on-entry，就要调用 cancel-debug-on-entry：

M-x cancel-debug-on-entry RET triangle-debugged RET

（如果你在 Info 中阅读这份文档，现在就可以取消 debug-on-entry。）

17.3 debug-on-quit 和 (debug)

除了设置 debug-on-error 和调用 debug-on-entry 之外，还有另外两种方法可以启动 debug。

通过将变量 debug-on-quit 设置为 t，可以使你无论何时键入 C-g(keyboard-quit)都能够启动 debug。这对于调试无限的循环很有用。

或者，你能够在你的代码中需要调试的一行中插入“(debug)”，就像这样：

```
(defun triangle-bugged (number)
  "Return sum of numbers 1 through NUMBER inclusive."
  (let ((total 0))
    (while (> number 0)
      (setq total (+ total number))
      (debug)                          ; Start debugger.
      (setq number (1= number)))       ; Error here.
    total))
```

有关debug 函数的详细描述，参见《*GNU Emacs Lisp* 技术手册》中的“Lisp 调试器”一节。

17.4 源代码级调试器 edebug

edebug 通常显示你所调试的函数的源代码。对于你当前执行的那一行，edebug 用一个箭头在左边进行提示。

你可以一行一行地跟踪整个函数的执行，或者快速运行直到到达一个断点处(在这个断点处 Emacs 执行停止)。

edebug 在《*GNU Emacs Lisp* 技术手册》中的“edebug”一节中有详细介绍。

这里是用于triangle-recursively 的一个带有 bug 的函数定义。对于triangle-recursively 函数定义本身，可以参见 11.2.2节“ 用递归算法代替计数器”。下面显示的这个例子没有对 defun 这一行进行通常的缩排。

```
(defun triangle-recursively-bugged (number)
  "Return sum of numbers 1 through NUMBER inclusive.
Uses recursion."
  (if (= number 1)
      1
    (+ number
       (triangle-recursively-bugged
        (1= number)))))                ; Error here.
```

通常，将光标置于函数定义的最后一个括号之后并键入 C-x C-e(eval-last-sexp)，或者将光标置于函数定义之中并键入 C-M-x(eval-defun)，你都能够安装这个函数。(默认情况下，eval-defun 命令仅仅工作在 Emacs Lisp 模式或者 Lisp 交互模式中。)

然而，要为 edebug 准备这个函数定义，你必须首先要用另外的命令将这个函数定义配置好。在 Emacs第19版中，将光标置于函数定义中并键入下面的命令就可以实现这个目的：

```
M-x edebug-defun RET
```

这个命令在没有加载 edebug 的情况下使 Emacs 自动地加载 edebug，然后正确地配置这个函数。(在加载 edebug 之后，你可以使用它的标准的绑定键，如C-u C-M-x (eval-defun以及一个前缀参量)，来调用 edebug-defun。)

在 Emacs 第18版中，需要自己来加载 edebug。将适当的 load 命令增加到你的“.emacs”初始化文件中就行了。

如果你在 Info 中阅读这份文档，你能够配置上面显示的这个 triangle-recursively-bugged 函数。对于函数定义行是缩进的那个函数定义，edebug-defun 无法正确定位其边界，因此这个例子没有按通常的缩排方式显示。

配置完函数之后，将光标置于下面的表达式后并键入C-x C-e(eval-last-sexp)：

```
(triangle-recursively-bugged 3)
```

光标将回退到 triangle-recursively-bugged 函数源代码中，而且光标位于这个函数的 if 表达式那一行的开始。同样，你将看到，一个箭头在这一行的左边，就像这样：“⇒”。这个箭头表示这一行是函数当前运行的那一行。

```
=>*(if (= number 1)
```

在这个例子中，位点的位置是用一个星号“*”显示(在 Info 中，位点则是以“-!-”显示)。

如果你现在键入空格键(SPC)，位点将移动到下一个执行的表达式，显示是这样的：

```
=>(if *(= number 1)
```

当你继续按下空格键的时候，位点将从一个表达式移动到下一个表达式。同时，每当表达式返回一个值时，这个值将显示在回显区中。例如，当你将位点移过 number 之后，你将看到下面的内容：

```
Result: 3 = C-c
```

这意味着 number 变量的值是 3，它对应着 ASCII码 的 C-c(CTL-c)。

你可以继续移过代码直到你到达有错误的那一行。在求值之前，这一行是：

```
=>          *(1= number)))))                    ; Error here.
```

当你按下空格键时，你将产生这样一个错误消息：

```
Symbol's function definition is void: 1=
```

这是程序的一个 bug。

按“q”键就可以退出 edebug。

要去除一个函数定义的 edebug，只要对它重新求值而不安装即可。例如，你可以将光标置于函数定义的最后一个括号之后并键入 C-x C-e 即可。

edebug 完成的工作比仅仅遍历整个函数代码多得多。你可以将它设置成自行运行，直到到达有错误的那一行或者在指定的断点才停下来，你可以使它显示表达式值的改变；你可以找出一个函数被调用了多少次，等等。

edebug 在《*GNU Emacs Lisp* 技术手册》中的“edebug”中有详细介绍。

17.5 调试练习

- 安装 count-words-region 函数，然后使之在你调用它的时候进入一个内置的调试器。在一个包含两个单词的区域运行这个命令。你将需要按下 d 键很多次。在你的系统中，在这个命令执行完之后，是否有一个“hook”调用？（关于 hook 的更多信息，参见《*GNU Emacs Lisp* 技术手册》中“命令循环概述”一节。）
- 将 count-words-region 函数拷贝到草稿缓冲区，如果需要，将defun一行前面的空格去掉，为 edebug 配置好这个函数，并跟踪它的执行。无需为这个函数制造一个 bug，虽然你完全可以做到这一点。如果一个函数没有 bug，遍历整个函数也是没有任何问题的。
- 在运行 edebug 的时候，键入“？”，看一看 edebug 的命令列表。（global-edebug-prefix 通常绑定到 C-x X，也就是 CTL-x 后接一个大写的 X，使用这个命令前缀可以暂时离开 edebug 的调试缓冲区。）
- 在 edebug 的调试缓冲区中，用 p (edebug-bounce-point) 命令看一看 count-words-region 函数在区域中的什么位置执行。
- 将位点移动到函数之后，然后键入 h (edebug-goto-here)命令，以使之跳到这个位置。
- 使用 t (edebug-trace-mode) 命令以使 edebug 自行跟踪这个函数的执行；使用大写的 T 用于 edebug-Trace-fast-mode，试一试，看结果会是什么？
- 设置一个断点，然后在跟踪模式中运行 edebug 直到它到达停止点。