

“C-c C-t”组合键查看不带文本内容的各级标题，请看下面的例子。我们将从刚才那个简单的大纲开始，先隐藏其正文，如下所示：

按下：C-c C-t

```
All About the Universe
*Preface| ...
**Scope of book...
**Intended audience...
*Chapter 1
**Universe basics
```

正文被隐藏起来，原来的两个正文行现在用省略号表示。

要想查看所有被隐藏起来的文本——不管它们是标题还是正文，请按下“C-c C-a”组合键（命令名是 **show-all**）。**hide-body** 和 **show-all** 这两个命令是惟一以整个大纲为操作对象的命令，其他命令（这些命令可就多了）则都是以光标所在的大纲部分为操作对象。用不着一口气把全部的命令都学会，先掌握几个最基本的，其余的可以到需要使用它们时再学。在大纲模式里做编辑工作的时候，使用**hide-body** 和 **show-all** 命令的时间占百分之九十以上。

可以用 **hide-subtree** 命令 [“C-c C-d”（注 4）] 组合键把某个标题的下级小标题和正文都隐藏起来。光标可以在这个标题上，也可以在其相关文本中的任何位置。接着以刚才的大纲为例子，准备把“Preface”分枝隐藏起来。如下所示：

按下：C-c C-d

```
All About the Universe
*Preface| ...
*Chapter 1
**Universe basics
```

省略号表示有被隐藏起来的文本。

注意屏幕画面的变化。发出 **hide-body** 命令的时候，原先的两个正文行不见了，取而代之的是两个省略号。发出 **hide-subtree** 命令的时候，大纲的某个分枝（包括其中的下级小标题）不见了，取而代之的是一个省略号。

注 4：如果使用的 Emacs 版本早于 19.25，这个命令默认的按键绑定是“C-c C-h”。

“**C-c C-s**”组合键（命令名是**show-subtree**）的作用正好与**hide-subtree**相反。如果光标在某个标题上并且该标题下有被隐藏起来的文本，“**C-c C-s**”组合键将把所有被隐藏起来的文本都显示出来，不管它们是正文还是下级小标题。

“**ESC x hide-entry RETURN**”命令的作用是，把紧跟在某个给定标题下的文本隐藏起来，别的下级小标题及其文本都不受影响。要想显示用**hide-entry**命令隐藏起来的文本，请发出“**ESC x show-entry RETURN**”命令。

文本被隐藏时的编辑

知道如何隐藏和显示文本之后，我们再来看看被隐藏文本的某些特点。有时候，需要对含有隐藏文本的文档进行编辑。这是一种调整文档布局结构的重要手段，但大家必须高度重视这类编辑工作可能带来的隐患。假设已经用大纲模式把全体正文都隐藏起来了，只有各级标题还显示在屏幕画面里。换句话说，此时看到的是文档的一份真正的“大纲”。如果移动了一个标题而这个标题带有隐藏着的小标题和相关文本，那么所有被隐藏起来的东西也会随着可见文本的移动而移动。稍后，当再次“显示”全篇文档的时候，刚才隐藏的文本将出现在新位置上——就在刚才移动的那个标题之下。类似地，如果删除了一个标题，它下面隐藏的文本也将被删除掉。

这个功能使段落的移动工作很方便。但有些事情大家必须注意：如果删除了某个标题后面的省略号，Emacs就会删除相关的被隐藏文本。为了避免误操作，Emacs会限制进行这样的操作——它不允许用**DEL**键来删除省略号，也不允许用“**C-b**”之类普通的光标移动命令把光标移动到省略号上。如果想用“**C-d**”命令删除省略号，那么Emacs会把隐藏文本重新显示到屏幕上，而不是一声不吭地删除它们。但如果就是想这样做或者因为运气不好，在编辑过程中因删除省略号（比如，使用了“**C-k**”命令）而丢失文本的事情还是会发生。虽然还可以用“**C-y**”命令把省略号恢复回来，但当重新显示文本的时候，有些东西可能已经丢失了。重新输入省略号并不能挽回丢失的信息，只能从自动保存(*auto-save*)文件（文件名是“#filename#”）或备份文件（文件名是“filename~”）里想法恢复它们。

对隐藏文本进行的编辑操作还有一个规定：如果想把隐藏着的文本移动到另外一个编辑缓冲区里，这两个编辑缓冲区就必须都在大纲模式下。如果试图把大纲里的隐藏文本移动到另外一个不在大纲模式下的编辑缓冲区里，就会看到来自大纲的文本



段落之间会夹杂着一些“^M”字符，有些文本甚至会丢失。如果把两个编辑缓冲区都放到大纲模式下，就能毫无问题地在这两个窗口之间移动文本。

最后，在文档里有文本被隐藏起来的同时，当然还可以添加新的标题和文本。但一定要小心，千万不能在省略号上进行输入，因为在省略号上的输入内容将会被插入到隐藏文本里。（即使能够预见到会发生什么样的事情，这种做法也不值得。）

使用大纲副模式

大纲模式也可以作为一个副编辑模式（minor mode）来使用。也就是说，完全可以在自己喜欢的主编辑模式（major mode）里使用大纲。以副编辑模式启动大纲模式的命令是“**ESC x outline-minor-mode RETURN**”，状态行上将出现“Outl”字样。从某些方面来说，这个编辑模式不是那么方便，大多数大纲命令前面的前缀将不再是一个简单的“C-e”。在大纲副模式下，必须给大纲命令都加上“C-e C-o”前缀，这样才能保证它们不会与主编辑模式里正常的“C-e”命令发生冲突。因此，如果想移动到下一个标题位置上去，就必须按下“**C-e C-o C-n**”组合键（相当于大纲模式里的“**C-e C-n**”命令）。

请注意，混合使用大纲主模式和大纲副模式不仅多余，而且有可能导致难以预料的后果。在某个编辑缓冲区进入大纲模式或者大纲副模式之后，就不要再画蛇添足了。在主模式里打开同功能的副模式（不仅限于大纲模式）反而会让 Emacs 不知所措。应该宁肯先保存文件、退出编辑缓冲区，然后再重新打开这个文件，也不应该让一个编辑缓冲区在大纲主模式和大纲副模式之间来回切换。

对大纲模式进行定制

现在，再来看看大纲模式的工作情况。（这部分内容比较晦涩难懂，如果你是一位 Emacs 新手，可以跳到下一小节去。）大纲模式由一个名为 **outline-regexp** 的变量控制，这个变量的值是一个正则表达式。这个正则表达式的作用是匹配任何一个标题行的开头部分。匹配到的字符串长度决定标题的级别。比如，变量 **outline-regexp** 的默认值是“*+”（它实际上会更复杂一些，但两者相差不多）。这个正则表达式匹配的是一个或者多个星号。大纲模式认为一颗星代表着一个一级标题，因为它匹配

到一个字符；两颗星代表着一个二级标题，因为它匹配到两个字符，依次类推。实际匹配到多少个字符并不重要，但更长的匹配永远代表着更低级别的标题。

如果各位足够聪明，就可以利用这个正则表达式让大纲模式适应自己的实际工作情况。请看下面这个正则表达式：

```
\(\(\(\(\.11\)\)\)\)|\(\(\(\.\le2\)\)\)\)|\(\(\(\.\lev3\)\)\)\|\(\(\(\.\level4\)\)\)
```

够乱的吧？这个正则表达式本可以写得更紧凑些，不过那样就更难懂了。它使用了将在第十三章介绍的几个知识点。就这个正则表达式而言，它将匹配以下格式的标题：

.11	一级标题
.le2	二级标题
.lev3	三级标题
.level4	四级标题

如果大家对 **troff** 比较熟悉，就会发现上面这些标题的格式与 **troff** 中的情况比较接近。现在，只需把上面这些标题转换为熟悉的 **troff** 标题宏（注 5），再把下面这条语句添加到 “*.emacs*” 文件里：

```
(setq outline-regexp "the-ugly-thing-you-just-saw")
```

就可以用大纲模式来处理正规的 **troff** 文本。**TeX** 用户也能构造出类似的东西。表 8-7 对大纲模式下的常用命令进行了汇总。

注 5：这个问题与本书内容无关，但完全可以这样做。它需要至少掌握一个 **troff** 小技巧：**troff** 软件里的宏名字只能有两个字符。但当它遇到类似于 “.levle4” 这样的东西时，它会把这个东西看做是对 “.le” 宏的一次调用，而 “vel4” 则是这次宏调用的第一个参数。也就是说，写一个名为 “.le” 的宏并让它测试其第一个参数是否是 “2”、“v3”、“vel4” 就可以达到目的。再顺便说一句，**troff** 的新版本（特别是自由软件基金会的 **groff**）已经允许使用两个以上的字符作为宏的名字。

表 8-7：大纲模式命令速查表

键盘操作	命令名称	动作
C-c C-n	outline-next-visible-heading	移动到下一个标题
C-c C-p	outline-previous-visible-heading	移动到上一个标题
C-c C-f	outline-forward-same-level	移动到下一个同级的标题
C-c C-b	outline-backward-same-level	移动到上一个同级的标题
C-c C-u	outline-up-heading	移动到上一级别的标题层次
C-c C-t	hide-body	隐藏全体正文行
C-c C-d	hide-subtree	隐藏某标题的全体下级小标题及其相关正文
(无)	hide-entry	隐藏某标题的正文部分(不包括它的下级小标题和那些小标题的正文部分)
C-c C-l	hide-leaves	隐藏某标题的正文部分,同时隐藏它全体下级小标题的正文部分(只留下标题)
C-c C-a	show-all	显示被隐藏起来的所有东西
C-c C-s	show-subtree	显示某给定标题的下级小标题和它的相关文本
(无)	show-entry	显示某给定标题的相关文本(不包括它的下级小标题及它们的正文部分)
C-c C-k	show-branches	显示某标题的正文及其全体下级小标题的全部正文
C-c C-i	show-children	显示某标题的下一级小标题(不包括正文文本)



第九章

用 Emacs 设置 排版标记

本章内容：

- 设置 troff 和 nroff 排版标记
- 设置 TeX 和 LATEX 排版标记
- 编写 HTML 文档
- Emacs 的 Html-helper 模式

上一章对 Emacs 能够实现的几种基本排版操作进行了介绍。要想用 Emacs 实现更专业的排版，就必须给文本加上一些特殊的标记——排版代码，然后用一个文本处理软件再对它进行处理。

nroff、**troff**、**TeX**和**LATEX**等文本处理软件会根据文本文件里的排版标记对文本进行处理，然后在激光打印机上产生一份打印输出。这本书就是用自由软件基金会提供的一种名为**groff**的工具软件（它是**troff**家族中的一员）生成的。

生成高质量的打印文稿是标记语言的用途之一，而 WWW 又为标记语言提供了另外一个广阔天地。程序员用 HTML (Hypertext Markup Language，超文本标记语言) 给 ASCII 文本加上排版标记，然后把文件放到 WWW 服务器上。世界各地的用户们再用一种名为浏览器的工具软件去浏览这些文件的内容。浏览器以加有排版标记的文件为输入，以专业效果（虽然有一些高低之分）的格式把它们显示在用户的显示器上，文件的显示效果取决于浏览器的能力。常见的 UNIX 浏览器包括 Mosaic、Netscape Navigator、Lynx 以及 Emacs 下的 W3 等等。

为这些种类繁多的文本和超文本处理系统创建标记文件并不困难，有标准配置的 Emacs 就足以完成这项工作了。这一章介绍的几个编辑模式能大大简化给文件添加排版标记这项相当琐碎和麻烦的工作，很值得学习。

有些字处理软件和其他工具把排版和编辑两种功能集成在一起。这类软件通常被称为“所见即所得”(WYSIWYG, 英文“what you see is what you get”的缩写)工具。但这类工具的文件格式往往不互相兼容。如果想把文件传递到另外一个系统上去、得先设法把它的排版标记去掉。Emacs 是一种 ASCII 文本编辑器, 它不是一个“所见即所得”工具。

在这一章里, 我们将探讨以下几种排版工具:

- **troff** 是 UNIX 世界传统的文本排版程序。**nroff** 的功能与 **troff** 相当, 主要用来对字符终端的输出进行排版, **man** 命令用的就是它。**groff** 来自由软件基金会, 与 **troff** 和 **nroff** 的功能相当。
- **T_EX** 这个排版软件的作者是 Donald Knuth, 主要用来排版图书, 它是我们将要介绍的文本处理软件中最精巧的一个。**L_AT_EX** 是一套由 Leslie Lamport 开发出来的 **T_EX** 命令。**T_EX** 和 **L_AT_EX** 可以从美国数学学会 (American Mathematical Society) 获得, 但需要支付发行费用。
- **Html-helper** 是 Emacs 中一种专门用来给 WWW 网上的文件添加排版标记的主编辑模式。编写的 HTML 文档可以用 Netscape 或者 Emacs 的 W3 等浏览器来查看, 关于 W3 的使用方法请参考第七章里的介绍。

Emacs 有各种能够帮助在文本里插入排版命令 [或者叫“标记 (markup) ”] 的主编辑模式。虽然 Emacs 提供的帮助有多少之分, 但使用专为文本排版工具而设计的编辑模式肯定会使工作更加顺畅。

注释

这一章所介绍的编辑模式与将在第十二章中介绍的 C 模式 (C mode) 和 FORTRAN 模式 (FORTRAN mode) 等程序设计语言编辑模式有一个共同的特点: 即都允许你在文档里使用注释, 而且都使用同一个键盘命令 “**ESC :**” (命令名是 **indent-for-comment**) 插入正确的注释语法。表 9-1 列出了本章各种排版工具的注释语法, 如下所示。



表 9-1：各种标记模式下的注释

如果在下列模式里敲入“ ESC ; ”	Emacs 将插入
nroff 模式	\”
T _E X 模式	%
L _A T _E X 模式	%
html-helper 模式	<!-- -->
	<!-- -->

设置 troff 和 nroff 排版标记

除了多出一些操作上的方便以外，nroff 模式与文本模式毫无二致。就将要探讨的问题来看，nroff 与它的近亲 troff 和 groff 之间没有任何差别；在这一节里介绍的命令对整个 troff 家族都适用。

要想进入 nroff 模式，请输入“**ESC x nroff-mode RETURN**”命令。状态行上将出现“Nroff”字样。nroff 模式提供以下 4 项功能：

- 它修改了 **paragraph-separate** 变量的正则表达式，重新定义了“段落”的含义。修改后的设置情况将保证 **fill-paragraph** 命令（“**ESC q**”组合键）能够正确地工作。
- 它提供了一些用来在 troff 文档里漫游的特殊命令。这些命令能够以文档中的 **troff** 命令为跳跃点做前后移动。
- 一个特殊的“nroff 配对模式”（electric nroff mode，一个与 nroff 模式相关联的副模式）能自动提供某些排版命令的收尾标记，确保它们能配对出现。
- 它提供了一些用来在文档中插入注释的操作命令。

段落的分隔与段落重排

我们将依次对上述 4 项功能进行介绍。先来看看“段落”的重新定义问题。如果经常做大量的文本编辑工作并用惯了“**ESC q**”组合键（命令名是 **fill-paragraph**），就一定要提防 troff 文档中的“段落”陷阱。Emacs 把空白行视为段落的结束标志，但 troff 文档却往往通篇没有一个空白行。因此，Emacs 将会因把整个文件当做一个

段落来进行重排而弄出大麻烦。[幸亏 Emacs 19 里的 **undo** 命令（“**C-x u**”组合键）的功能很强，足以恢复这种混乱的局面。]

troff 排版命令（通常被称为“请求（request）”或者“宏（macro）”）必须从左页边开始输入，而且必须以一个句点（.）或单引号（'）打头。不管 **fill-paragraph** 命令干了些什么，它也不能改动这些行。因此，除了普通的段落定义（以空白行分隔的文字块）之外，nroff 模式还将把以一个句点，或一个单引号打头的文本行定义为一个段落。举个例子，请看下图中的文本：

初始状态：

```
Buffers Files Tools Edit Search Help
.H1 "Various Ways of Questioning about the Thing"
.H2 "Philosophical and Scientific Questioning"
.PP
From the range of the basic questions
of metaphysics we shall
here ask this
.I
one
.R
question:What is a thing?
.PP
-----Emacs:heidegger      (Nroff Fill)--L5--All-----
```

这是一些为 **troff** 准备的文本。要想对第一段（“From the range”）进行重排，请把光标移动到这个段落的中间并按下 “**ESC q**” 组合键。如果是在其他编辑模式里，这个命令会把整个文件重排为一个很长的段落。但在 nroff 模式里，“**ESC q**” 组合键将只对两个宏定义标记之间的一小段文本进行重排。

按下： **ESC q**

```
Buffers Files Tools Edit Search Help
.H1 "Various Ways of Questioning about the Thing"
.H2 "Philosophical and Scientific Questioning"
.PP
From the range of the basic questions of metaphysics we shall here ask
this
.I
one
.R
question:What is a thing?
.PP
-----Emacs:heidegger      (Nroff Fill)--L4--All-----
```

Emacs 对文本进行段落重排，只有夹在两个宏定义标记之间的那个段落发生了变化。

如上所示，Emacs 没有重排整个文档，甚至没有重排整个段落，它只对夹在“.PP”宏定义标记和“.I”宏定义标记之间的那一小部分文本进行了重排。

在文档中移动

如果工作文件里散布着很多 troff 排版命令，想在其中从某个地方移动到其他地方有时候会很让人着急，必须跳过文件里的宏定义标记，可它们好像总拦在路上。nroff 模式提供了一种简便的方法，使得能够跳过两个文本行之间的一切宏定义标记从上一行直接移动到下一行。“**ESC n**”组合键将把光标移动到下一个文本行，“**ESC p**”组合键将把光标移动到上一个文本行。如图 9-1 所示。

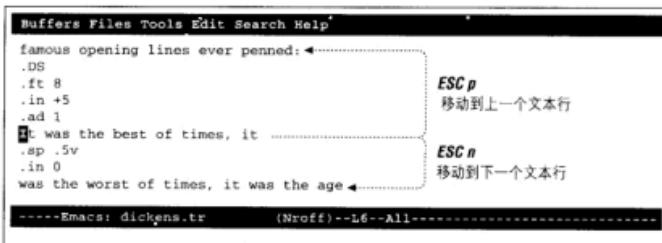


图 9-1: nroff 模式中的移动

排版标记要单独占用一个文本行，所以标记文件看上去好像加长了很多。nroff 模式专门提供了一个命令来统计“真正”的文本行有多少。“**ESC ?**”命令就是用来统计某个给定的文本块里（不包括排版标记占用的那些行）到底有多少行文本的。如果想知道文件里总共有多少个文本行，可以先用“**C-x h**”组合键把整个编辑缓冲区选取为一个文本块，再用“**ESC ?**”命令统计出其中的文本行个数。（用来创建各种标题的标记语句将包括到文本行的统计当中，因为它们将以文本行的面目出现在打印稿里。）

分节符的定义和使用

重新定义 Emacs 的 **page-delimiter** 变量，以便能够对 troff 的分节符进行查找费不了多大的事，但对工作却会有很大的帮助。这个变量是一个正则表达式，它对页面的

分隔符进行了定义；诸如“**C-x]**”（命令名是**forward-page**）之类的命令就是靠**page-delimiter** 变量来判断下一页是从什么地方开始的。在默认的情况下，变量 **page-delimiter** 查找的是换页符（即“**C-l**”字符），但可以重新设置这个变量，让它查找**troff** 的分节符。

举个例子。假设正在使用 **ms** 宏定义包。用 **ms** 宏定义包设置的文档会使用宏定义标记“**.NH**”来开始一个新的节（**section**）。于是，把 **page-delimiter** 变量设置为“**^ \\.NH**”就等于是告诉 **Emacs** 出现在某行开头的“**.NH**”标记表示将从这里开始一个新的打印页。做了这样的设置之后，就可以用翻页命令在 **troff** 文档的节之间移动了。

要想对 **page-delimiter** 变量做这样的设置，请输入“**ESC x set-variable RETURN page-delimiter RETURN “^ \\.NH” RETURN**”命令。如果想让这个设置固定下来，就需要把下面这条语句添加到“**.emacs**”文件里：

```
(setq nroff-mode-hook '(lambda () (setq page-delimiter "^\\"\\.\"NH\"")))
```

如果使用的宏定义包不是 **ms**，就得自己想法写出 **page-delimiter** 变量用的正则表达式。对正则表达式的详细介绍请参考第十三章。

注意：**page-delimiter** 变量并不是一项 **nroff** 模式独有的功能。不管编辑哪类文档，都可以使用类似的技巧。

宏定义标记的配对出现

nroff 配对模式（Electric **nroff** mode）又多提供了一项功能。**troff** 的宏定义标记经常是配对出现的，一个起始标记对应着一个配对的收尾标记。比如“**.TS**”表示某个表格从此处开始，而“**.TE**”则表示这个表格到此处结束。在 **nroff** 配对模式里，**Emacs** 会自动插入一组必须配对出现的宏定义标记中的收尾标记，只需输入其中的起始标记即可。

要想进入 **nroff** 配对模式，首先要进入 **nroff** 模式，然后输入“**ESC x electric-nroff-mode**”命令。事实上，**nroff** 配对模式只是一个给“正宗的” **nroff** 模式增加了新功能的副模式，本身不是一个独立的主模式。因此，不能直接进入 **nroff** 配对模式。

进入nroff配对模式之后，再遇到需要输入一组必须配对出现的宏定义标记时，只需先输入其中的起始标记再按下“C-j”组合键（或LINEFEED）即可。Emacs会自动插入其收尾标记，并且会在它们之间插入一个空自行；然后，Emacs将把光标放在这两个宏定义标记之间的空白行上。不用移动光标就可以继续输入文本——这种安排特别适合那些喜欢一边写作一边设置排版标记的人们使用。下面的画面给出了用nroff配对模式输入一组需要配对出现的宏定义标记“.DS”和“.DE”的情况：

输入：.DS



输入一组必须配对出现的两个命令中的起始命令。

按下：C-j



Emacs插入与之配对的结束命令，并把光标放在它们的中间。

nroff配对模式可以自动补足由-ms、-me和-mm等宏定义包所定义的各组配对标记，一些通常配对出现的 troff 指令也包括在内。表 9-2 列出了 nroff 配对模式能够识别和自动补足的各组必须配对出现的 troff 宏定义标记。

表 9-2：nroff 配对模式支持的配对命令组

起始标记 (由用户输入)	收尾标记 (由 Emacs 自动插入)	使用情况
.(b	.)b	me 宏定义包
.(l	.)l	me 宏定义包
.(c	.)c	me 宏定义包

表 9-2: nroff 配对模式支持的配对命令组 (续)

起始标记 (由用户输入)	收尾标记 (由 Emacs 自动插入)	使用情况
.(x	.)x	me 宏定义包
.(z	.)z	me 宏定义包
.(d	.)d	me 宏定义包
.(q	.)q	me 宏定义包
.(f	.)f	me 宏定义包
.LG	.NL	ms 宏定义包
.LD	.DE	ms 宏定义包
.CD	.DE	ms 宏定义包
.BD	.DE	ms 宏定义包
.DS	.DE	ms 宏定义包
.DF	.DE	ms 宏定义包
.FS	.FE	ms, mm 宏定义包
.KS	.KE	ms 宏定义包
.KF	.KE	ms 宏定义包
.LB	.LE	mm 宏定义包
.AL	.LE	mm 宏定义包
.BL	.LE	mm 宏定义包
.DL	.LE	mm 宏定义包
.ML	.LE	mm 宏定义包
.RL	.LE	mm 宏定义包
.VL	.LE	mm 宏定义包
.RS	.RE	ms 宏定义包
.TS	.TE	tbl 预处理器
.EQ	.EN	eqn 预处理器
.PS	.PE	pic 预处理器
.BS	.BE	mm 宏定义包
.na	.ad b	troff 指令
.nf	.fi	troff 指令
.de	..	troff 指令

nroff 模式中的注释

下面再来看看 nroff 模式里的注释是如何工作的。如果出现在文本行行首的 3 个字符是 “. \ ”，那么这一行就将被整个看做是一句注释（注释不会出现在打印稿里）。如果在文本行的半中间出现了 “\ ” 这两个字符，则它们后面的内容就会被看做是注释。就我们的使用经验看，nroff 模式中的注释似乎有些小问题，但大家最好还是应该练习一下这项功能的使用——别看它有毛病，你可能会发现它很有用呢。

假设已经启用了自动换行模式（auto-fill mode，这是一个副模式）。然后在输入过程中，决定在文本行上插入一句注释。Emacs 监测到输入了一个注释标记（\ ）。当输入的东西超出了右页边时，Emacs 将自动开始一个新行；但它会认为在新行上输入的东西还是注释。为了让工作更顺畅，Emacs 将把注释标记 “\ ” 自动插入到新行的行首。Emacs 只会在新行的行首自动插入注释标记。如果不在自动换行模式里，或者是按下了回车键，那么它就不会做任何特殊的事情。

遗憾的是，Emacs 漏掉了那个打头的英文句号。这将导致 troff 在打印输出里额外留出一个空白行。除了回过头去动手插入那个英文句号外，我们不知道还有什么其他明显的方法能够纠正这一问题。也许定义一个宏（让它使用一个正则表达式查找与替换操作）来自动纠正插入在行首处的注释标记会更简单一些。（我们把它留给大家作为一个练习，详细情况请参考第十章。）

命令 “**ESC ;**”（命令名是 **indent-for-comment**）能够在当前行的末尾自动插入一个注释标记（\ ）。如果这一行上没有别的东西，那么 **indent-for-comment** 命令的行为将非常奇怪：第一次按下 “**ESC ;**” 组合键的时候，它会在行首插入 “\ ” 这两个字符；第二次按下 “**ESC ;**” 组合键的时候，它会把刚才的注释标记替换为 “. \ ”（注意：在句点和反斜线之间有一个空格）；第三次按下 “**ESC ;**” 组合键的时候，它才会变为 “. \ ”。也就是说，如果想在一个空白行上设置一个传统的 nroff 注释标记，就必须在该空白行上连接三次 “**ESC ;**” 组合键。如果不喜欢反复多次地按这个组合键，可以把它绑定为一个一次就能解决问题的组合键。

把 nroff 模式设置为默认的主编辑模式

如果需要频繁使用 troff，那么把 nroff 模式设置为默认的主编辑模式可能更方便一些。nroff 模式完全可以看作额外多出几条命令的文本模式，它几乎能胜任各项工作。

作。如果想让 nroff 模式成为默认的主编辑模式, 请把下面这条语句添加到 “.emacs” 文件的最开始:

```
(setq default-major-mode 'nroff-mode)
```

如果 “.emacs” 文件里有把默认的主编辑模式设置为其他模式 (比如文本模式) 的语句, 记得要删除它。为了让这条新语句生效, 需要保存这个文件并重新进入 Emacs。表 9-3 对 nroff 模式里的各种命令进行了汇总。

表 9-3: nroff 模式命令速查表

键盘操作	命令名称	功能
(无)	nroff-mode	进入 nroff 模式
ESC n	forward-text-line	把光标移动到下一个文本行
ESC p	backward-text-line	把光标移动到上一个文本行
ESC ?	count-text-lines	统计文本块中的文本行数
(无)	electric-nroff-mode	进入一个副编辑模式。在这个模式里, 输入必须配对出现的 nroff 命令组中的第一个, 按 “C-j” 组合键, Emacs 就会自动插入该配对命令组中的第二个命令
C-j	electric-nroff-newline	只能用在 nroff 配对模式里, 自动插入一组必须配对出现的 troff 宏定义标记中的第二个
ESC ;	indent-for-comment	在文本里插入一个注释标记

设置 TeX 和 LATEX 排版标记

GNU Emacs 对 TeX 文件的标记支持功能, 比它对 troff 文件的标记支持功能更全面。TeX 模式和 LATEX 模式都是主编辑模式, 只要启用了它们, 各种专用操作就会生效, 帮助用户在 Emacs 里对这些文件进行标记和排版处理。TeX 模式的许多功能也能用在 LATEX 模式里。LATEX 用户应该在学完这一小节之后, 再去学习在 LATEX 模式里增加的其他功能。

进入 TeX 模式的方法是输入 “**ESC x tex-mode RETURN**” 命令。Emacs 将检查文件是否像是一个 TeX 或 LATEX 文件, 并进入正确的编辑模式里。如果想特别指定 TeX

模式, 请输入 “**ESC x plain-tex-mode RETURN**” 命令。类似地, 如果想启动 LATEX 模式, 请输入 “**ESC x latex-mode RETURN**” 命令。

括号的配对出现

TeX命令的常见格式是 “`\keyword{text}`”。TeX模式不会特别注意所用的关键字是否“正确”, 这是因为 TeX 本身是可扩展的, 允许用户定义自己的关键字。不过, 它确实提供了一种用来避免出现常见 TeX 错误的功能, 如花括号 () 和美圆符号 (\$) 不配对。

在 TeX 里, 花括号 () 和美圆符号 (\$) 永远要配对出现; Emacs 会检查每个左花括号和美圆符号是否都有对应的配对符号。输入一个右花括号或美圆符号时, 光标将迅速移动到与之对应的配对符号处 (前提是配对符号也在屏幕上)。如果配对符号不在屏幕上, 它就会把左、右括号之间的上下文显示在辅助输入区里, 然后再回到原处。这种可视性的检查工作对光标在文件中的位置没有影响。如果在光标移向左括号的同时继续输入, 那么输入的文本将出现在正确的位置上, 它不会把字符输入到错误的地方 (注 1)。

花括号经常会多个嵌套在一起, 所以光标可能不会移动到距离最近的左括号处, 而是移动到与这个右括号相对应的那个左括号处。举个例子, 请看下面这个嵌套着 3 重花括号的文本行:

```
\keyword{this is some \boo{funny text with \bar{many} nested braces}
          1           2           3   3   2   1
```

即使找不到配对的括号, Emacs 也不会报警; 而且, Emacs 也无法肯定它找到的括号是否正是那个“正确”的配对括号。用户必须用自己的眼睛去观察, 这样才能确保 Emacs 找到了另一半括号, 而且它找到的那个括号是正确的。Emacs 也不会在整个编辑缓冲区的范围里去搜索配对括号——这样做太花时间了。它只向回搜索一

注 1: “向回移动到配对符号的起始元素处”是 Emacs 在遇到圆括号和方括号时的标准动作 (这一问题的详细探讨请参考第十二章)。尽管圆括号和方括号与 TeX 的排版工作没有什么瓜葛, Emacs 也同样会在这种编辑模式里检查圆括号和方括号的配对情况。因为 Emacs 自己的出错信息通常是笼统地说“括号不配对”, 所以此后的讨论中, 我们也将经常用“括号”一词来笼统地表示花括号、美圆符号和圆括号。

个有限的距离，如果找不到，就放弃搜索。因此，如果配对括号远在文件的其他部位，在输入右括号时可能就看不到左括号。

Emacs 能够帮助生成配对的括号。命令 “C-c {” 将在文本里插入一对左、右括号，然后把指示打字位置的输入光标放到两个括号的中间。如下所示：

按下： C-c {

```
Buffers Files Tools Edit Search TeX Help
Blah, blah, blah blah[

--***-Emacs: texblahs          (TeX Fill)--L1--All--
```

“C-c {” 组合键插入一对花括号。

按下 “C-c }” 组合键会把光标移过右括号。它总是能根据当前位置找到正确的右括号。请看下面这个例子。假设光标在单词 “funny” 上时按下了 “C-c }” 组合键，光标将移动到对应的右括号（即单词 “nested” 后面的那个右括号）的后面。如下所示：

按下： C-c }

```
Buffers Files Tools Edit Search TeX Help
\keyword{this is some \boo{funny
text with \bar{lots of} nested} braces}

--***-Emacs: funnies          (TeX Fill)--L2--All--
C-c }
```

“C-c }” 组合键跳出花括号的当前层次。

如果光标原来在单词 “boo” 上或者更靠前的地方，“C-c }” 组合键将把光标移动到第二行的第一个括号的后面。如果 Emacs 没有找到配对的右括号，就将显示一条 “unbalanced parentheses (括号不配对)” 的出错信息。如果光标所在的段落并没有被括号包围起来而你却按下了 “C-c }” 组合键，就也会看到这条出错信息——这多少有点容易引起误会。

要想检查是否有不配对的花括号和圆圈符号，请输入 “**ESC x validate-tex-buffer RETURN**” 命令。这个命令会在整个编辑缓冲区的范围内检查是否有尚未配对的圆

括号、花括号、美圆符号或者其他类似的符号。如果找到这类错误，Emacs 将打开一个“*Occur*”编辑缓冲区，其顶部显示“Matches:(不配对)”字样，然后是 Emacs 找到的出错文本行的清单。可以利用“**ESC x goto-line**”命令迅速到达出现错误的每一行。

有时候，编辑缓冲区里一个比较靠前的不配对括号会因连锁反应，而造成文件里出现大量“错误”的局面。如果认为某个改正也会因连锁反应，而把剩余的大部分错误都“改正”过来，可以随时再执行一次 **validate-tex-buffer** 命令。

逐个改正错误的时候，刚才介绍的“**C-e }**”组合键可以帮助查明与某给定左括号正确配对的右括号到底在什么地方，只需你把光标放在左括号的后面再按下“**C-e }**”组合键即可。

引号和段落分隔

TeX模式还有一项专门对引号和段落分隔符进行处理的功能。在TeX或LaTeX模式里输入双引号字符（"）将使 Emacs 插入左双引号（“）和右双引号（”）。这些字符其实是两个用来模拟左双引号和右双引号的单引号；左、右双引号不是 ASCII 字符集里的标准字符。如果确实需要输入一个双引号（"），可以在双引号字符前加上一个字符引用命令，即按下“**C-q "**”组合键（命令名是 **tex-insert-quote**）。

按“**C-j**”（或者 **LINEFEED**）组合键可以正确插入TeX和LaTeX的段落分隔符——两个 **RETURN** 字符。它同时检查前一段落里是否存在有不配对的括号。比如，如果在按下“**C-j**”组合键的时候，前一个段落里还有个不配对的左括号——即少了一个右括号，那么 Emacs 会报告：

```
Paragraph being closed appears to contain a mismatch
```

这种警告可以让用户一边写作一边对段落进行检查，用不着等到最后再对整个编辑缓冲区做全面检查，这有助于做到有错即纠。

注释

在TeX模式里，按下“**ESC ;**”组合键将在当前行的末尾插入一个注释标记（一个百分号 "%")。

文本的排版和打印

在不退出 Emacs 的前提下，不仅能够给文件添加 TeX 排版标记，还可以对它们实际进行处理并查改设置错的排版标记。用来对整个编辑缓冲区进行排版处理的键盘动作是按下“**C-c C-b**”组合键（命令名是 **tex-buffer**）。原本应该显示到屏幕上的信息将被引导到一个名为“*tex shell*”的编辑缓冲区，Emacs 会把这个编辑缓冲区显示在屏幕上。如果没有在屏幕上看到这个编辑缓冲区，请按下“**C-c C-l**”组合键（命令名是 **tex-recenter-output-buffer**），这将使它自动显示在屏幕上。

对 TeX 编辑缓冲区的部分内容进行排版处理需要一个特殊的技巧。TeX 文件有一个专门用来完成各种初始化设置工作的文件头，随后才是包含文本和排版标记的正文部分。因此，为了对编辑缓冲区的部分内容进行排版处理，Emacs 先得找到文件头，然后先把它发送给 TeX 排版软件，再把选取的文本块（即准备对之进行排版处理的那部分文件内容）发送过去。为了让 Emacs 能够找到文件头，需要把文件头放在下面这两个字符串之间：

```
%**start of header  
%**end of header
```

上面两个字符串都是以百分号（%）打头的，而这个符号在 TeX 里是注释语句的语法，所以这两行语句不会影响文件的排版处理工作。

请看下面这个例子。对一个典型的 TeX 文件来说，文件头将对文件里用到的各种关键字（比如字体设置等）进行定义，随后才是加有排版标记的文本。在插入上面这两个字符串之后，整个文档的布局应该是下面这样的：

```
%**start of header:  
TeX header stuff  
%**end of header  
Regular text  
The region you want to print  
More regular text
```

对 TeX 文件来说，这两个字符串会告诉 Emacs 文件头在什么地方。在打算打印的文本块的一端设置上文本块标记，再把光标移动到文本块的另一端，然后按下“**C-c C-r**”组合键（命令名是 **tex-region**）。如果在排版工作完成之前又想取消它，请按下“**C-c C-k**”组合键（命令名是 **tex-kill-job**）。

用这些排版命令处理 TeX (或 LATEX) 文件会生成一个 “.dvi” 文件，这是一种独立于具体打印设备的过渡性文件；如果想把它打印出来，还需要把它再进一步转换为打印机的执行命令。要想把 “.dvi” 文件打印出来，需要使用 “C-c C-p” 组合键（命令名是 **tex-print**）；它将完成对 “.dvi” 文件的转换并把它送到默认打印机去。“C-c C-q”（命令名是 **tex-show-print-queue**）组合键可以把打印队列显示出来，这样用户就能知道自己该在什么时候去打印机那里查看排版后的打印输出稿了。

有几个重要的变量会告诉 Emacs 如何来打印一份 TeX 文件。如果 “C-c C-p” 或者 “C-c C-q” 组合键工作不正常，就得检查它们的设置情况；如果这些命令根本就不工作，那么系统上的 TeX 软件的配置可能不合标准，或者是打印命令、打印队列命令与普通情况稍有差异。变量 **tex-command** 负责确定用来运行 TeX 软件的命令，它的默认值在 TeX 模式里是 “tex”，在 LATEX 模式里是 “latex”。变量 **tex-dvi-print-command** 负责确定用来打印 “.dvi” 文件的命令；它的默认值是 “lpr -d”。再来看看打印队列、用来显示打印队列的命令是由变量 **tex-show-queue** 控制的。在默认的情况下，**tex-show-queue** 变量被设置为 “lpq”。这些默认设置都是比较合理的，但 TeX 的配置情况变化非常大。因此，如果各位的 TeX 模式需要进行一些定制才能正确工作，请不必感到惊讶。

LATEX 模式与 TeX 模式的差异

LATEX 是 TeX 的一种简化版，推出 LATEX 的目的是想用 TeX 软件模仿商用软件 Scribe。对 TeX 模式的介绍全部适用于 LATEX 模式，所以在学习这一小节之前，应该先看看介绍 TeX 模式的上一节内容。LATEX 模式多了一项功能：可以用 “C-c C-e” 组合键插入一组配对命令中的第二个命令。请看下图。

输入：\begin{document} RETURN C-c C-e



如果输入 LATEX 一组配对排版命令中的第一个命令后再按下 “C-c C-e” 组合键，Emacs 将自动插入这组配对命令中的第二个命令。

“**C-c C-e**”组合键适用于任何关键字。因为 LATEX (类似于 TeX) 是可定制的，所以 Emacs 无法判断使用的关键字是不是一个有效的 LATEX 关键字，甚至无法判断是不是对它进行过定义。举个例子，如果输入的是 “**\begin{eating} C-c C-e**”，Emacs 就会插入 “**\end{eating}**”。使用的关键字是否合法有效责任全在于用户自己。

如果想打印输出某个 LATEX 编辑缓冲区中的部分内容，则不必给文件头加上任何特殊的标记。LATEX 文件头格式相当统一，所以 Emacs 不需要任何帮助就能找出文件头的起始和结束位置。

表 9-4 对 TeX 和 LATEX 命令进行了汇总。

表 9-4：TeX 和 LATEX 命令速查表

键盘操作	命令名称	功能
(无)	tex-mode	根据文件内容进入 TeX 模式或 LATEX 模式
(无)	plain-tex-mode	进入 TeX 模式
(无)	latex-mode	进入 LATEX 模式
C-j	tex-terminate-paragraph	插入两个硬回车 (标准的段落结束标记) 并检查段落的语法
C-c {	tex-insert-braces	插入两个花括号，然后把光标放在它们中间
C-c }	up-list	如果光标在两个花括号中间，就把光标移到紧跟右括号的后面
(无) <i>TeX → Validate Buffer</i>	validate-tex-buffer	检查编辑缓冲区有无语法错误
C-c C-b <i>TeX → TeX Buffer</i>	tex-buffer	对编辑缓冲区进行 TeX 或 LATEX 排版处理
C-c C-l <i>TeX → TeX Recenter</i>	tex-recenter-output-buffer	把排版信息画面显示在屏幕上。 (至少) 会显示最后一条出错信息
C-c C-r <i>TeX → TeX Region</i>	tex-region	对文本块进行 TeX 或 LATEX 排版处理
C-c C-k <i>TeX → TeX Kill</i>	tex-kill-job	中断 TeX 或 LATEX 的排版处理工作

表9-4: \TeX 和 \LaTeX 命令速查表(续)

键盘操作	命令名称	功能
C-c C-p $\text{\TeX} \rightarrow \text{\TeX Print}$	tex-print	打印 \TeX 或 \LaTeX 的排版输出
C-c C-q $\text{\TeX} \rightarrow \text{Show Print Queue}$	tex-show-print-queue	查看打印队列
C-c C-e	tex-close-latex-block	只能用在 \LaTeX 模式里, 自动插入某个命令组中的第二个命令

编写HTML文档

随着WWW的日益流行,许多人正在用HTML语言给文档加上Web格式的排版标记。HTML文档的内容都是些简单的ASCII文本,但里面加有各种用来定义文本显示特性的标签(tag)。HTML文档写起来并不困难,Emacs自己就已经足够写HTML标签和文本用的了。HTML标签通常采用下面这样的格式:

```
<tagname>text being tagged</tagname>
```

在Emacs里编写HTML还有几个增值模式可供使用,包括html模式、html-helper模式、html菜单编辑模式,以及包括SGML模式和psgml模式等在内的各种SGML工具等。(HTML是SGML的一个具体实现。SGML是Standard Generalized Markup Language(标准化通用标记语言)的缩写。)我们将从这么多的工具里挑出html-helper模式介绍给大家。

Html-helper模式允许输入任何标签,不管它属于HTML 2.0、HTML 3.0还是由某第三方创造出来的一个扩展。但也正是由于它这种兼收并蓄的特性,html-helper模式不适合用来检查HTML文档的语法错误。检查HTML文档最好的方法是用几种流行的浏览器来浏览它们。

我们并不打算在以下各个小节里教大家如何去编写HTML。(关于编写HTML的进一步资料,请参考《HTML: The Definitive Guide》一书,它的作者是Chuck Musciano和Bill Kennedy,由O'Reilly & Associates出版公司出版。)我们的目的是把使用html-helper模式编写HTML文档的基本方法介绍给大家。



Emacs 的 Html-helper 模式

Html-helper 模式是由 Nelson Minar 开发的，它能以非常灵活的方式编写 HTML；根据不同的经验和个人爱好，在启用或者禁用各种帮助功能方面，每个用户有充分的选择余地。它提供了大量用来输入 HTML 配对标签的键盘命令；而且，如果给它们加上修饰（办法是先按下“**C-u**”组合键），它们就能自动把标签加到文本块的两端而不是把它们放到光标位置上。作为键盘命令的一种替代品，html-helper 模式还支持自动补足功能。只需输入标签的头几个字符，再按下“**ESC TAB**”组合键（命令名是 **tempo-complete-tag**）就能把它输入到文档里。如果可用的选择不止一个，那么 Emacs 会把各种可能的补足情况列在一个窗口里以供挑选。

Html-helper 模式不包括在 Emacs 的默认配置里，但可以从因特网下载，它的 URL 是 <ftp://ftp.reed.edu/pub/src/html-helper-mode.tar.gz>。（如果你对下面的内容看不懂，请向你的系统管理员求助，注 2。）找到这个 tar 文件后，先把它下载到某个目录里，例如 “`~/elisp`”，然后进入那个目录并输入以下命令：

```
% gunzip html-helper-mode.tar.gz  
% tar xvf html-helper-mode.tar
```

系统将对 tar 文件进行解包操作。这个 tar 文件包含有以下几个组件：

- *html-helper-mode.el* —— 对应于 html-helper 模式的 LISP 文件
- *tempo.el* —— 允许插入各种 HTML 文档模板的 Lisp 文件
- *key bindings.html* —— 键盘命令清单
- *changelog.html* —— 关于升级和改进方面的日志记录
- *configuring.html* —— 关于配置 html-helper 模式的指示
- *differences.html* —— 介绍 HTML 模式和 html-helper 模式之间的差异
- *documentation.html* —— 关于 html-helper 模式的文档

注 2： 如果打算安装与 Emacs 一起工作的 LISP 软件包，请系统管理员帮忙通常不是个坏主意。他也许会把那个软件包安装到系统上的某个中央地点，这可以让更多的人来访问和使用它。

启动 Html-helper 模式

要想启动 html-helper 模式, 请输入 “**ESC x load-file RETURN**”, Emacs 会询问想加载哪个文件, 输入 *html-helper-mode.el* 或 *html-helper-mode.elc* 文件的完整路径名作为回答, 然后按回车键。接着, 输入 “**ESC x html-helper-mode**” 命令; 状态行上将出现 “HTML helper” 字样。

把html-helper模式设置为Emacs启动工作的一部分也很简单。把下面这些语句添加到 “*.emacs*” 文件里即可:

```
(setq load-path (cons 'PUT_THE_PATH_HERE load-path))
(setq auto-mode-alist (cons ('html$ . html-helper-mode) auto-mode-alist))
```

请把存放 *html-helper-mode.el* 文件的子目录的完整路径, 放到第一条语句中的引号里。第二条语句的作用是让 Emacs 在启动 Emacs 时, 自动加载 html-helper 模式。第三条语句告诉 Emacs: 只要是准备编辑一个以 “.html” 为后缀名的文件, 它就自动进入 html-helper 模式。

特殊字符的输入

在 HTML 里, 尖角括号 (“<” 和 “>”) 和 “&” 都是特殊字符。html-helper 模式允许直接输入特殊符号, 但如果想在 HTML 文档里显示 “<”、“>” 或 “&” 字符, 就必须输入与之对应的转义代码才行。在输入字符之前, 先按下 “**C-c**” 组合键就可以输入与该字符对应的转义代码。比如, 按下 “**C-c <**” 组合键将输入对应于小于号的转义代码 (<)。

HTML 文档框架

Html-helper 模式为 HTML 初学者准备了一个很有用的功能, 即能选择让 Emacs 在创建一个新 HTML 文件时自动插入一个文档框架。(有经验的老手也许会因已经自行开发出一套模板文件而用不着这项功能了。) 如果想让 html-helper 模式自动插入一个文档框架, 请把下面这条语句添加到 “*.emacs*” 文件里:

```
(setq html-helper-build-new-buffer t)
```

保存 “*.emacs*” 文件并重新启动 Emacs。然后来创建一个新的 HTML 文档。

输入：C-x C-f *new.html*

```
Buffers Files Tools Edit Search Help
<html> <head>
<title></title>
</head>

<body>
<h1></h1>

<hr>
<address></address>
<!--hhmts start -->
<!--hhmts end -->
</body> </html>

-----Emacs: new.html      (HTML helper)--L1--All-----
Loading html-helper-mode...done
```

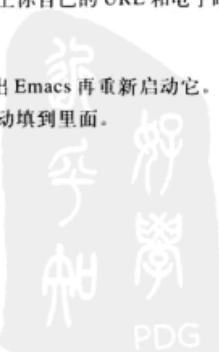
HTML-helper 模式插入一个文档框架，编写一个 HTML 文档所需要的基本元素都已经准备齐全。

HTML 的基本元素都包含在这个框架里。整个文档被括在 `<html>` `</html>` 标签的中间。随后是标题部分和正文部分。再往后的那个 `<hr>` 标签告诉浏览器在此插入一条水平线，这条线路通常被称为水平标尺。`<address>` 标签给文档的作者留出一块用来填写其因特网地址的地方。如果把下面这句语句添加到 “*.emacs*” 文件里，HTML helper 模式就能把这个地址自动添加到今后创建的每一个 HTML 文档里：

```
(setq html-helper-address-string
      '<a href= \"http://www.merryold.co.uk/~chuck/\">Charles Dickens &lt;
      chuck@merryold.co.uk&gt;</a>')
```

当然，如果你不是 Charles Dickens，就得在这里填上你自己的 URL 和电子邮件地址。

保存 “*.emacs*” 文件，但不保存此编辑缓冲区；退出 Emacs 再重新启动它。现在，当打开 *new.html* 文件的时候，Emacs 就会把地址自动填到里面。



输入：C-x C-f new.html

```

Buffers Files Tools Edit Search Help
<html> <head>
<title></title>
</head>

<body>
<h1></h1>

<hr>
<address><a href="http://www.merryold.co.uk/~chuck/">Charles Dickens
&lt;chuck@merryold.co.uk&gt;</a></address>
<!-- hhmts start -->
<!-- hhmts end -->
</body> </html>

--***-Emacs: new.html          (HTML helper)--L1--All-----

```

Emacs 插入的 HTML 文档框架，里面已经填上了地址。

开始写作。在文档框架里放上一个文档标题（title）和一个一级标题（header）（它们经常是一模一样的），然后开始输入文字内容。在输入一段文字之后，按“**ESC RETURN**”插入一个HTML段落分隔符——这是很少几个不需要配对的标签之一，因此只需输入一个就行。比如，先把狄更斯小说《双城记》（A Tale of Two Cities）的第一段输入进去，然后按下“**ESC RETURN**”组合键。请看下图。

先输入狄更斯小说《双城记》第一段，然后按下“**ESC RETURN**”组合键。

```

Buffers Files Tools Edit Search Help
<html> <head> <title>A Tale of Two Cities</title> </head> <body> <h1>A
Tale of Two Cities</h1> It was the best of times, it was the worst of
times, it was the age of wisdom, it was the age of foolishness, it was
the epoch of belief, it was the epoch of incredulity, it was the season
of Light, it was the season of Darkness, it was the spring of hope, it
was the winter of despair, we had everything before us, we had nothing
before us, we were all going direct to Heaven, we were all going direct
the other way—in short, the period was so far like the present period,
that some of its noisiest authorities insisted on its being received,
for good or evil, in the superlative degree of comparison only.
<p> | <hr>
--***-Emacs: dickens.html          (HTML helper)--L16--Top-----

```

Emacs 插入一个段落分隔符“<p>”。

在文本块的首尾配对设置 HTML 标记

在编写 HTML 的时候，对现有文件进行编码往往要多于真正的写作。如果在输入任何一个标签之前按下了“**C-u**”组合键，那么 Emacs 会把标签自动放到一个文本块的两端，而不是把它们插入到光标的当前位置。

请看下面的例子。我们准备给 Strunk 和 White 两人合著的《Elements of Style》书中的某些文字加上一些标记——这些文字输入到一个文件里（注 3）。打开一个名为 *strunk.html* 的新文件，Emacs 会自动插入一个文档框架。把光标移动到一级标题的标签下方，准备插入未经标记排版的 ASCII 文本文件 *strunk*。

输入：C-x C-f *strunk.html*

```
Buffers Files Tools Edit Search Help
<html> <head>
<title></title>
</head>
|
<body>
<h1></h1>
|
<hr>
<address><a href="http://www.strunk.com/~white/*">E.B. White
&lt;white@strunk.com&gt;
</a></address>
<!-- hhmts start -->
<!-- hhmts end -->
</body> </html>
---- Emacs:strunk.html      (HTML helper)--L1--All-----
```

Emacs 自动插入一个 HTML 文档框架。

把光标移到配对出现 *<h1>* 标签组的后面，输入：C-x C-i *strunk*

```
Buffers Files Tools Edit Search Help
<html> <head>
<title></title>
</head>
|
<body>
<h1></h1>
Elementary Rules of Usage
|
1. Form the possessive singular of nouns by adding 's.
Follow this rule whatever the final consonant. Thus write,
|
Charles's friend
Burns's poem
the witch's malice
---- Emacs: strunk.html      (HTML helper)--L7--Top-----
```

Emacs 插入 ASCII 文本文件 *strunk*，我们将给它添加一些 HTML 标签。

注 3：当然，我们只是用这段文字来练习编写 Web 页面，而不是要把它张贴到因特网上，因为它受版权保护。

在这个例子里，我们打算让文档标题（title）和一级标题（header）不一样。

把光标移动到<title>和</title>之间，然后输入：The Element of Style, Chapter 1

```
Buffers Files Tools Edit Search Help
<html> <head>
<title>The Elements of Style, Chapter 1 </title>
</head>

<body>
<h1></h1>
Elementary Rules of Usage

1. Form the possessive singular of nouns by adding 's.

Follow this rule whatever the final consonant. Thus write,

Charles's friend
Burns's poem
the witch's malice
---***Emacs: strunk.html      (HTML helper)--L2--Top---
```

文档标题输入完毕。

把“Elementary Rules of Usage”选取为一个文本块，按“C-w”组合键把它剪切下来，然后用“C-y”组合键把它插到标签<h1>和</h1>的中间。

```
Buffers Files Tools Edit Search Help
<html> <head>
<title>The Elements of Style, Chapter 1</title>
</head>

<body>
<h1>Elementary Rules of Usage </h1>

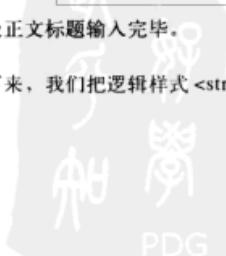
1. Form the possessive singular of nouns by adding 's.

Follow this rule whatever the final consonant. Thus write,

Charles's friend
Burns's poem
the witch's malice
---***Emacs: strunk.html      (HTML helper)--L6--Top---
```

一级正文标题输入完毕。

接下来，我们把逻辑样式用到第一个句子上。



把第一个句子选取为一个文本块，然后按下“**C-u C-c C-s s**”组合键（命令名是**tempo-template-html-strong**）。

```

Buffers Files Tools Edit Search Help
<html><head>
<title>The Elements of Style, Chapter 1</title>
</head>

<body>
<h1>Elementary Rules of Usage</h1>

<strong>1. Form the possessive singular of nouns by adding 's.</strong>
Follow this rule whatever the final consonant. Thus write,
Charles's friend
Burns's poem
the witch's malice
--**-Emacs: strunk.html      (HTML helper)--L8--Top-----

```

Emacs 把 `` 和 `` 标签分别插到文本块的两端。

接下来，按一次回车键，然后用“**ESC RETURN**”组合键插入一个段落分隔符。再移动到随后的第二段文字的后面，并再次按下“**ESC RETURN**”组合键。

输入：**C-n ESC RETURN C-d C-n ESC RETURN**

```

Buffers Files Tools Edit Search Help
<html><head>
<title>The Elements of Style, Chapter 1</title>
</head>

<body>
<h1>Elementary Rules of Usage</h1>

<strong>1. Form the possessive singular of nouns by adding 's.</strong>
<p>
Follow this rule whatever the final consonant. Thus write,
<p>
Charles's friend
Burns's poem
the witch's malice
--**-Emacs: strunk.html      (HTML helper)--L12--Top-----

```

Emacs 将插入两个段落分隔符。

我们接下来将对从“Charles's friend”开始的几行文本进行标记。我们准备把它们安排在一个项目符号列表（bulleted list）里。

把示范文字选取为一个文本块，然后输入：**C-u C-c C-l u**



```
Buffers Files Tools Edit Search Help
<html><head>
<title>The Elements of Style, Chapter 1</title>
</head>

<body>
<h1>Elementary Rules of Usage</h1>

<strong>1. Form the possessive singular of nouns by adding 's.</strong>
<p>
Follow this rule whatever the final consonant. Thus write,
<p>
<ul>
    <li>Charles's friend
Burns's poem
the witch's malice

</ul>
-----Emacs: strunk.html      (HTML helper)--L17--Top-----
```

Emacs 将插入项目符号列表（即无序列表，unordered list）的起始标签和结束标签。

对列表的处理差不多已经完成了，但我们还想再对它做一些细致的调整。把光标移到单词“Burns”的字母“B”上，按下“**C-c C-l i**”组合键（命令名是**html-helper-smart-insert-item**，之所以称它为“smart insert（智能插入）”是因为标签能够用来对项目符号列表、编号列表以及菜单等进行标记）。Emacs 插入标签，再对“the witch's malice”做同样的设置。注意**html-helper**模式会自动把列表缩进一点点。如果把列表嵌套在另外一个列表里，它的缩进量还会更大一些。因为空自行和缩进对 HTML 文档的显示效果并没有什么影响，所以这里的缩进只是为了提高 HTML 文件的可读性而已。如果想禁用缩进功能，请把下面这条语句添加到“**.emacs**”文件里：

```
(setq html-helper-never-indent 't)
```

如果还想继续对这个文件进行标记，可以用同样的方法给其余文字都加上 HTML 标签。在插入 HTML 标签的时候，既可以使用各种键盘命令，也可以使用自动补足功能。我们下面就来说说自动补足功能。

Html-helper 模式下的自动补足功能

Html-helper模式里的键盘命令实在是太多了；根本不可能把它们都记下来。好在我们还有另外一种选择——输入前几个字符之后按下“**ESC TAB**”组合键，Emacs 或者自动补足HTML标签，或者显示一个窗口把各种可用的补足形式都列在其中以供选择（如果鼠标可以用，那么就可以用鼠标中键挑选一个补足形式）。自动补足功能在 Emacs 19.23 及其以后的版本里都能工作。

假设还在 strunk.html 文件的文本上工作。已经完成了下一个段落的标记，现在想再设置一个项目符号列表。如下所示：

输入：ESC TAB

```
Buffers Files Tools Edit Search Help
</ul>
Exceptions are the possessives of ancient proper names in -es and -is,
the possessive of Jesus', and such forms as for conscience' sake, for
righteousness' sake. But such forms as Moses' laws, Isis' temple are
commonly replaced by
<p>

<ul>
  <li>
</ul>the law of Moses
the temple of Isis
--**-Emacs: strunk.html      (HTML helper)--L10--65%
```

Emacs 插入了列表的首、尾标签和一个列表数据项标签。

在上面的例子里，必须把列表中的数据项剪粘到适当的位置。自动补足功能无法对文本块进行标记，但很适用于先输入标签的一部分，再由 Emacs 补足它的情况。

如果你选择亲自动手输入HTML标签，就将发现Emacs会以在大多数其他编辑模式里对圆括号和方括号进行配对检查的类似方法，对html-helper模式里对尖角括号进行配对检查。当你输入右尖角括号之后，Emacs将把光标移动到与之对应的左尖角括号处，然后再回到那个右尖角括号，使你能够对尖角括号的配对情况进行视觉上的检查。

启用输入提示功能

有些HTML标签需要输入信息域中的内容。比如在输入超链接的时候，就必须输入

该链接的URL地址和用来让用户选取该超链接的文本。如果用按下“**C-c C-a l**”（小写的字母“L”）组合键的方法输入一个链接，那么Html-helper模式将自动插入如下所示的标签：

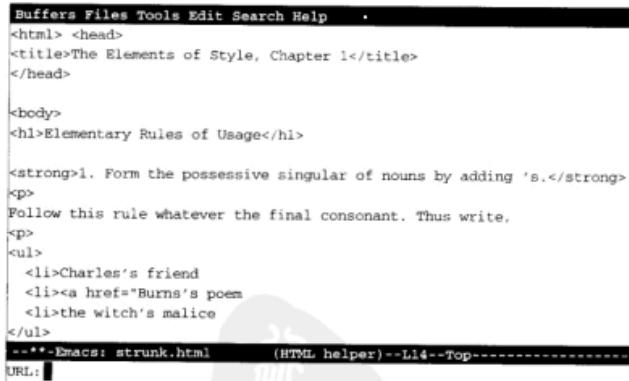
```
(a href=" " </a>)
```

并把光标放到第二个双引号上，于是可以直接输入URL（网址）。但要是启用了提示功能，html-helper模式还可以让这个标记操作更简便。启用提示功能的办法是把下面这条语句添加到“*.emacs*”文件里：

```
(setq tempo-interactive t)
```

请看下面的例子。假设打算在*strunk.html*文档里放上一个指向某个介绍Robert Burns生平事迹的Web主页的（假想的）超链接，我们来看看输入提示功能是如何工作的。

把“Burns's”选取为一个文本块，然后按下“**C-u C-c C-a l**”（小写的字母“L”）组合键：



The screenshot shows the Emacs interface with the following content:

```
Buffers Files Tools Edit Search Help
<html> <head>
<title>The Elements of Style, Chapter 1</title>
</head>

<body>
<h1>Elementary Rules of Usage</h1>

<strong>1. Form the possessive singular of nouns by adding 's.</strong>
<p>
Follow this rule whatever the final consonant. Thus write,
<p>
<ul>
    <li>Charles's friend
    <li><a href="Burns's poem
        <li>the witch's malice
    </ul>
-- Emacs: strunk.html      (HTML helper)--L14--Top--URL: [ ]
```

Emacs 提示输入超链接的 URL（网址）。

输入: <http://www.uofillyria.edu/poets/RobertBurns.html> RETURN

```
Buffers Files Tools Edit Search Help
<html> <head>
<title>The Elements of Style, Chapter 1</title>
</head>

<body>
<h1>Elementary Rules of Usage</h1>

<strong>l. Form the possessive singular of nouns by adding 's.</strong>
<p>
Follow this rule whatever the final consonant. Thus write,
<p>
<ul>
<li> Charles's friend
<li> <a href="http://www.uofillyria.edu/poets/
    RobertBurns.html">Burns </a> his poem
<li> the witch's malice
</ul>
--- Emacs: strunk.html (HTML helper)--L4--Top---
```

Emacs把这个超链接插入到文档里。

如果还没有输入超链接本身的文字内容, Emacs也同样会提示输入它。(因为我们已经把超链接本身的文字内容选取为一个文本块, 又给标记命令加上了“C-u”前缀, 所以Emacs会把超链接插入到文本的两端。掌握给现有文本增加超链接的办法是很必要的, 因为大多数作者都是用“先写作、后加超链接”的办法逐步对文档进行“超链化”。)

如果在启用输入提示功能的情况下输入列表, Html-helper模式将提示输入列表中的数据项。下面, 我们一起用编号列表标记命令(“C-c C-l o”组合键)来给 Emacs 列一个优点清单, 这次使用的文件叫做 *emacs.html*。如下所示:

输入: C-x C-f *emacs.html*, 然后移过 *<h1> </h1>*, 再按下“C-c C-l o”:

```
Buffers Files Tools Edit Search Help
<html> <head>
<title></title>
</head>

<body>
<h1></h1>
<ol>
<li> [ ]
</ol>
--- Emacs: emacs.html (HTML helper)--L8--All---
```

Emacs 提示输入一个列表数据项。

输入: Emacs is extensible RETURN



A screenshot of the Emacs interface. The menu bar at the top includes 'Buffers', 'Files', 'Tools', 'Edit', 'Search', and 'Help'. Below the menu bar is a buffer containing the following HTML code:

```
<html> <head>
<title></title>
</head>

<body>
<h1></h1>
<ol>
  <li> Emacs is extensible
</ol>
```

The cursor is positioned after the word "Emacs". At the bottom of the screen, there is a status line with the text "---- Emacs: emacs.html (HTML helper) --L8-- All-----".

Emacs 插入列表的第一个数据项。

输入提示功能到底是有用还是瞎操心全在于个人。对于 HTML 新手，输入提示功能可以帮助做到把每个标签的必要信息都输入齐全。如果不喜欢它，那么只要把添加到 ".emacs" 文件里的那条语句删掉即可。

时间戳的自动刷新

时间戳 (timestamp) 的作用是告诉你以及浏览你主页的人们，所看到的信息是在什么时候做的最后一次更新。这需要每编辑一次文档就在其中放上一个时间戳，但可能会忘记这件事。不过，如果把下面这条语句添加到 ".emacs" 文件里，那么 html-helper 模式就会在编辑和保存某个 HTML 文件时自动插入和刷新时间戳：

```
(setq html-helper-do-write-file-hooks t)
```

以后每次保存文件时都能观察到时间戳的变化。现在就可以按下 “C-x C-s” 组合键试试，时间戳将改变为当前的日期和时刻。退出 Emacs 时不会看到时间戳所发生的变化，但它确实会发生变化。如果只是在浏览这个文件，或者编辑之后没存盘，Emacs 也不会刷新时间戳。如果刚开始编辑一个新的页面，则它的时间戳部分将是空白的。如下所示：



输入：**C-x C-f ralph.html**

```
Buffers Files Tools Edit Search Help
<html> <head>
<title></title>
</head>

<body>
<h1></h1>

<hr>
<address><a href="http://www.homestyle.com/ralph/">Ralph Radisson
&lt;ralph@homestyle.com&gt;</a></address>
<!-- hhmts start -->
<!-- hhmts end -->
</body> </html>

--*- Emacs: ralph.html          (HTML helper)--L1--All-----
```

Emacs 插入一个 HTML 文档框架，里面带着一个空白的时间截。

在这个文件里写了一些东西并对它进行了存盘之后，html-helper 模式就将自动对时间截进行刷新。

按下：**C-x C-s**

```
Buffers Files Tops Edit Search Help
<html> <head>
<title>Ralph's Homestyle Home Page</title>
</head>
<body>
<h1>Ralph's Homestyle Home Page</h1>

<hr>

Hello and welcome to my Homestyle Home Page, where you can learn
the secrets of my famous Homestyle Hamburger Hash,
<address><a href="http://www.homestyle.com/ralph/">Ralph Radisson
&lt;ralph@homestyle.com&gt;</a></address>
<!-- hhmts start -->
Last modified: Fri Dec 2 10:55:03 1996
<!-- hhmts end -->
</body></html>

--*- Emacs: ralph.html          (HTML helper)--L1--All-----
```

Emacs 会在每次文件存盘时自动刷新这个时间截。

表 9-5 列出了 html-helper 模式里的各种键盘命令。有些键盘命令是为表单 (form) 或 HTML 3.0 中的某些高级功能而准备的。有些标签在一般情况下会出现在不同的行上 (比如某些用来标记列表的标签);但在下面这份表格里,它们都显示在同一行上。

表 9-5: Html-helper 模式命令速查表

键盘操作	命令名称	动作
C-u	universal-argument	当加在其他标记命令的前面时,把标签插入到文本块的两端
C-e >	tempo-template-html-greater-than	插入大于号的转义代码 “>”
C-e <	tempo-template-html-less-than	插入小于号的转义代码 “<”
ESC RETURN	tempo-template-html-paragraph	插入段落分隔符 “<p>”
C-e C-s s	tempo-template-html-strong	插入 标签
C-e C-s e	tempo-template-html-emphasized	插入 标签
C-e C-s b	tempo-template-html-blockquote	插入 <blockquote> </blockquote> 标签
C-e C-s p	tempo-template-html-preformatted	插入 <pre> </pre> 标签
C-e C-l u	tempo-template-html-unordered-list	插入 标签
C-e C-l o	tempo-template-html-ordered-list	插入 标签
C-e C-l t	tempo-template-html-definition-item	插入 <dt> <dd> 标签
C-e C-l l	tempo-template-html-item	插入 标签
C-e C-l d	tempo-template-html-definition-list	插入 <dl> <dt> <dd> </dl> 标签
C-e C-l m	tempo-template-html-menu	插入 <menu> </menu> 标签
C-e C-l r	tempo-template-html-directory	插入 <dir> </dir> 标签
C-e C-l i	html-helper-smart-insert-item	插入 标签
ESC TAB	tempo-complete-tag	补足当前标签
C-e RETURN	tempo-template-html-break	插入一个换行符 “ ”
C-e -	tempo-template-html-horizontal-rule	插入一个水平标尺 “<hr>”

表 9-5: Html-helper 模式命令速查表 (续)

键盘操作	命令名称	动作
C-c SPC	tempo-template-html-nonbreaking-space	插入一个空格字符的转义代码“ ”并设定不得在此处断开文本行
C-c &	tempo-template-html-ampersand	插入一个“&”字符的转义代码“&”
ESC C-t	html-helper-insert-timestamp-delimiter-at-point	插入时间戳分隔符
C-c C-b t	tempo-template-html-title	插入 <title> </title> 标签
C-c C-a l	tempo-template-html-anchor	插入 标签
C-c C-a n	tempo-template-html-target-anchor	插入 标签
C-c C-t 6	tempo-template-html-header-6	插入 <h6> </h6> 标签
C-c C-t 5	tempo-template-html-header-5	插入 <h5> </h5> 标签
C-c C-t 4	tempo-template-html-header-4	插入 <h4> </h4> 标签
C-c C-t 3	tempo-template-html-header-3	插入 <h3> </h3> 标签
C-c C-t 2	tempo-template-html-header-2	插入 <h2> </h2> 标签
C-c C-t 1	tempo-template-html-header-1	插入 <h1> </h1> 标签
C-c TAB e	tempo-template-html-align-alt-image	插入 标签
C-c TAB i	tempo-template-html-image	插入 标签
C-c TAB t	tempo-template-html-alt-image	插入 标签
C-c TAB a	tempo-template-html-align-image	插入 标签
C-c C-f p	tempo-template-html-input-textarea	插入 <textarea name =""rows =""cols =""> </textarea> 标签
C-c C-f x	tempo-template-html-input-reset	插入 <input type = "RESET" value = ""> 标签
C-c C-f b	tempo-template-html-input-submit	插入 <input type = "SUBMIT" value = ""> 标签

表 9-5: Html-helper 模式命令速查表 (续)

键盘操作	命令名称	动作
C-c C-f a	tempo-template-html-input-audio	插入 <input type = "AUDIO" name = ""> 标签
C-c C-f s	tempo-template-html-input-scribble	插入 <input type = "SCRIBBLE" name = "" size = ""> 标签
C-c C-f g	tempo-template-html-input-image	插入 <input type = "IMAGE" name = "" src = ""> 标签
C-c C-f r	tempo-template-html-input-radio	插入 <input type = "RADIO" name = ""> 标签
C-c C-f c	tempo-template-html-input-select	插入 <select name = ""> 标签
C-c C-f u	tempo-template-html-input-url	插入 <input type = "URL" name = "" size = ""> 标签
C-c C-f d	tempo-template-html-input-date	插入 <input type = "DATE" name = "" size = ""> 标签
C-c C-f .	tempo-template-html-input-float	插入 <input type = "FLOAT" name = "" size = ""> 标签
C-c C-f i	tempo-template-html-input-int	插入 <input type = "INT" name = "" size = ""> 标签
C-c C-f t	tempo-template-html-input-text	插入 <input name = "" size = ""> 标签
C-c C-f f	tempo-template-html-form	插入 <form action = ""> </form> 标签
C-c C-n m	tempo-template-html-margin	插入 <margin> </margin> 标签
C-c C-n f	tempo-template-html-footnote	插入 <footnote> </footnote> 标签
C-c C-n n	tempo-template-html-note	插入 <note role = ""> </note> 标签
C-c C-n a	tempo-template-html-abstract	插入 <abstract> </abstract> 标签
C-c C-p r	tempo-template-html-render	插入 <render tag = "" style = ""> 标签
C-c C-p -	tempo-template-html-subscript	插入 标签
C-c C-p ^	tempo-template-html-superscript	插入 标签
C-c C-p x	tempo-template-html-strikethru	插入 <s> </s> 标签
C-c C-p f	tempo-template-html-fixed	插入 <tt> </tt> 标签



表 9-5: Html-helper 模式命令速查表 (续)

键盘操作	命令名称	动作
C-c C-p u	tempo-template-html-underline	插入 <u> </u> 标签
C-c C-p i	tempo-template-html-italic	插入 <i> </i> 标签
C-c C-p b	tempo-template-html-bold	插入 标签
C-c C-s a	tempo-template-html-address	插入 <address> </address> 标签
C-c C-s l	tempo-template-html-lit	插入 <lit> </lit> 标签
C-c C-s g	tempo-template-html-arg	插入 <arg> </arg> 标签
C-c C-s m	tempo-template-html-cmd	插入 <cmd> </cmd> 标签
C-c C-s .	tempo-template-html-abbrev	插入 <abbrev> </abbrev> 标签
C-c C-s y	tempo-template-html-acronym	插入 <acronym> </acronym> 标签
C-c C-s n	tempo-template-html-person	插入 <person> </person> 标签
C-c C-s q	tempo-template-html-quote	插入 <q> </q> 标签
C-c C-s d	tempo-template-html-definition	插入 <dfn> </dfn> 标签
C-c C-s v	tempo-template-html-variable	插入 <var> </var> 标签
C-c C-s k	tempo-template-html-keyboard	插入 <kbd> </kbd> 标签
C-c C-s r	tempo-template-html-citation	插入 <cite> </cite> 标签
C-c C-s x	tempo-template-html-sample	插入 <samp> </samp> 标签
C-c C-s c	tempo-template-html-code	插入 <code> </code> 标签
C-c C-b b	tempo-template-html-base	插入 <base href = ""> 标签
C-c C-b l	tempo-template-html-link	插入 <link href = ""> 标签
C-c C-b n	tempo-template-html-nextid	插入 <nextid> 标签
C-c C-b i	tempo-template-html-isindex	插入 <isindex> 标签

这个表格就已经够长的了 (有那么多的键盘命令), 可每天仍有许多的新 HTML 标签创造出来。插入新标签的方法之一是创造自己的命名宏 (named macro)。第十章对如何编写宏命令以进一步简化操作方面的问题进行了探讨。



第十章

Emacs 中的宏

本章内容：

- 什么是宏
- 定义宏
- 向现有的宏里增加编辑命令
- 命名并保存宏
- 执行一个已命名的宏
- 建立复杂的宏
- LISP 函数——宏的补充

什么是宏

什么是宏 (macro)？在 Emacs 里，“宏”是一组被录制下来的按键动作，可以反复多次地使用。有了宏，就可以从重复性的工作中解放出来。比如，假设想删除某个表格的第三栏。普通的方法是：先移动到第一行，再移动到第三栏，删除它；然后移动到第二行，重复执行完全相同的一组命令；如此周而复始直到全部完成。即使手不累，可心早就烦了。Emacs 允许对表格第一行进行处理时所使用的按键动作录制下来，然后重复“播放”这些动作直到整个工作完成（注 1）。

在 Emacs 里使用的任何命令或者动作，从输入文本到编辑再到切换编辑缓冲区，都可以在一个宏里完成。要想把宏用足用好，其中的关键是能否准确地判断出自己在什么场合会做重复性的工作。也就是说，在什么情况下会连续多次以同一顺序重复按下同一组按键。一旦学会判断什么是重复性工作，就能对应该在什么时候使用宏有一个良好的感觉。接下来，还需要有另外一种天赋，那就是在识别出一组“几乎完全相同”的按键动作之后，如何使它们到达“完全相同”的地步，即发掘出这样一组按键动作：如果重复使用它们，它们就会完全按照意愿达到目标。这两种技巧学习起来并不是特别困难，只要稍加练习，就能够随时享受宏带来的方便。

注 1：当然，正如第八章所描述的，通过把表格的第三栏标记为矩形，也可以删除第三栏。但让我们感到欣慰的是：当你发现自己正在做一些重复性工作时，宏是一个：“记忆”工具。

这听起来好像是一种“懒人的”编程思路，而事实也确是如此：宏能够用一种简单的方法来完成复杂的工作，而你则既不需要学习LISP程序设计语言，又不需要额外学习特殊的定制技巧。如果你设计的宏正好适用于必须频繁接触的某些工作，甚至可以把它们保存起来，然后在需要它们的时候再随时加载。这样，你可以创造出一系列非常方便的宏，让它们成为编辑文件时的专用命令。你可以不受Emacs提供的命令的限制创造出自己的东西来！

在 Emacs 里的工作情况决定了使用宏的目的。我们主要用宏：

- 给文本设置排版标记。
- 把某个编辑缓冲区里的标题作为一份大纲，复制到另外一个编辑缓冲区里。
- 做一些查询 - 替换无法应付的查找加替换类型的操作。
- 建立索引目录。
- 重新对从另外一个应用软件里导入进来的文件进行排版。
- 把表格从一种格式套用为另外一种格式。
- 用一个命令来完成某个程序的编译、运行和测试。

只要学会几个基本的宏操作命令，就能在想出它更多的用途。

定义宏

要想开始一个宏的定义工作，请按下“**C-x (**”组合键。状态行上将出现“Def”字样，表示现在是在宏定义模式（macro definition）里。在这个模式里，Emacs 将把所有的按键动作记录下来（不管它们是命令还是文字性的文本），从而可以在今后反复地使用。要想结束一个宏的录制工作，请按下“**C-x)**”组合键，离开宏定义模式。而 Emacs 也将停止记录按键动作。如果在宏的录制过程中出现了错误，或者如果按下了“**C-g**”组合键，Emacs 也会停止录宏的录制。

录制宏的时候，Emacs 会在记录按键动作的同时执行它们，这就是说，录制宏时输入的任何内容，都将被看做是一个正常的命令并被执行。定义宏时的操作与平时的编辑工作完全一样，以便看清录制中的宏是不是符合自己心愿；如果发现正在录制

的宏与想让它完成的工作不太一致，则可以随时取消它（按下“**C-g**”组合键）。令人遗憾的是，要想在Emacs里对宏进行编辑和修改是非常困难的。如果真的出了错，那么惟一现实的选择是取消这个宏，然后再重新开始另外录制一个（注2）。

要想执行已经录制好的宏，需要按下“**C-x e**”组合键（命令名是**call-last-kbd-macro**）；Emacs将把刚才录制下来的按键动作精确地重演一遍。在任一时刻只能有一个活动的宏。如果又定义了另外一个宏，那么新定义的宏就将成为活动的宏，而早先定义的那个将被它覆盖。

示例

请看下面定义和执行宏的演示示例。我们想在打印稿里使用双倍行距，就标准的ASCII文件而言，要想打印一份这样的文件在UNIX下是没有现成的办法可用的。下面这个宏能够把编辑缓冲区里的文本全部设置为双倍行距。

键盘操作	动作
C-x (开始录制一个宏；状态行上将出现“Def”字样
C-e	移动到当前行的行尾
C-f	移动到下一行的第一个字符
RETURN	插入一个空白行
C-f	向右移动一个字符（到达下一行的行首）
C-x)	结束宏的录制工作

大家可以找个文件副本来自试这个宏。

注2：这种说法并不很正确。如果能用**undo**命令撤销刚才犯的错误，那么完全可以删除并重新输入几个字符，或者移动一下光标，然后再继续录制宏——就像什么事情也没有发生过一样。宏执行起来非常快，所以几个额外的命令不会对它构成损害。但这样的宏往往存在着一些潜在的问题。记住：只有能够反复使用的宏才是真正有价值的。

按上表给出的按键动作定义一个宏。

```
Buffers Files Tools Edit Search Help
It was the best of times, it was the worst of times, it was the age of
wisdom, it was the age of foolishness, it was the epoch of belief,
it was the epoch of incredulity, it was the season of Light, it was the
season of Darkness, it was the spring of hope, it was the winter of
despair, we had everything before us, we had nothing before us, we were
all going direct to Heaven, we were all going direct the other way-in
short, the period was so far like the present period, that some of its
noisiest authorities insisted on its being received, for good or evil,
in the superlative degree of comparison only. It was the best of times,
it was the worst of times, it
--- Emacs: dickens (Text Fill)--L3--All---
```

在录制这个宏的过程中，对第1行进行了双倍行距处理，结束时光标留在了第2行上。

让我们“勇敢地”假设这个宏没有毛病，接下来让它重复执行4次——给用来执行一个宏的命令加上“C-u”前缀就能做到这一点。当然，在实际工作中，在如此大胆地放手操作之前至少应该先试一次。

输入： C-u C-x e

```
Buffers Files Tools Edit Search Help
It was the best of times, it was the worst of times, it was the age of
wisdom, it was the age of foolishness, it was the epoch of belief, it
was the epoch of incredulity, it was the season of Light, it was the
season of Darkness, it was the spring of hope, it was the winter of
despair, we had everything before us, we had nothing before us, we
were all going direct to Heaven, we were all going direct the
other way-in short, the period was so far like the present period, that
some of its noisiest authorities insisted on its being received, for good
or evil, in the superlative degree of comparison only.
--- Emacs: dickens (Text Fill)--L11--All---
C-u C-x e
```

现在对前5行完成了双倍行距的设置工作：1个是在录制宏时完成的，其余4个是通过执行那个宏而完成的。

宏的工作情况很理想；于是我们可以自信地把编辑缓冲区里剩余的文本行，都设置

为双倍行距：先输入几个“**C-u**”，然后输入“**C-x e**”（注 3）。当然，如果要更精确一些，可以先用“**ESC x count-lines-region RETURN**”命令找出准备设置双倍行距的区域里到底有多少行文本。比如，如果只想给当前段落设置双倍行距，而**count-lines-region**命令告知该段落只剩下 6 行还需要设置，那么只要给出“**ESC 6 C-x e**”命令即可。

下面是一些容易犯的错误，请大家注意避免：

- 在结束宏的录制工作时别忘了按下“**C-x)**”组合键。如果在宏的录制工作结束之前就试图执行之，那么 Emacs 会报警，而已经录制的击键动作也都将丢失。
- “**C-g**”组合键将终止宏编辑命令的录制工作，并且会使 Emacs 忘记已经录制的击键动作。
- 任何错误都将自动终止一个宏的执行。如果 Emacs 蜂鸣报警，就得重新开始。
- Emacs 将“毫无理智”地严格按照录制下来的击键动作来执行。因此，一定要避免作诸如“我在执行这个宏的时候肯定是在行首（或者行尾）”之类的假设。

如果执行了一个宏却做错了事，可以用“**C-x u**”组合键（正常的 undo 命令）来撤销。Emacs 会知道此时的“撤销上一条命令”操作的真正含义是“撤销整个宏”而不是“撤销宏里的最后一条命令”。

技巧：如何创建良好的宏

录制和重复使用击键动作的技巧并不难掌握。但新手往往会犯一些典型的错误：录制了一个宏，使用它，然后发现它的执行情况和预想并不是百分之百地吻合。其实只要稍加注意，就可以让自己的宏更有用，错误更少。

所谓良好的宏是那些能够在任何需要场合都能正确工作的宏。因此，应该在宏里使用绝对型命令而非相对型命令。比如，如果正在写一个用来把某排版字符串放到光标所在单词两端的宏，那么肯定希望这个宏不管单词本身有多长都应该能够工作。

注 3：为什么要先输入几个“**C-u**”呢？因为宏会在其执行过程中遇到错误时报警并停下来。而在这个例子里，如果光标已经到达文件尾，那么再向前移动一个字符就会产生一个“错误”。因此，哪怕设置双倍行距的那个文件只有 25 行，用“**C-u C-u C-u C-u C-x e**”命令来重复执行这个宏 256 次也是不会有问题的。

因此，应该使用像“**ESC f**”（命令名是 **forward-word**）之类的绝对型命令，而不是几个每次右移一个字符的“**C-f**”组合键。类似地，如果想移动到行首或者行尾，就应该使用“**C-e**”和“**C-a**”之类的命令，而不使用让光标做前后相对移动的命令。

宏通常都是以一个查找命令开始的，它的作用是进入要在文件里开始执行这个宏的地方。对于这种情况，输入查找参数（比如“**C-s searchstring**”）要比简单地使用“重复上一次查找”命令（如“**C-s C-s**”）效果更好。这是因为，在录制宏时，使用的查找字符串与打算执行那个宏时，所需要使用的查找字符串极有可能是不同的，而“**C-s C-s**”只记得住上一次查找操作所使用的查找字符串。

有时候，给宏额外增加几个并非严格必要的命令（通常是“**C-a**”、“**C-e**”）是很有好处的，它们的作用是确保操作发生在文本行上的正确位置。宏在录制时做的假设越少，它在执行时取得的效果往往也就越好。因此，如果一组命令只有在行尾时才能正确执行，那么宏就应该以“**C-e**”命令开始，哪怕已经“知道”“只有在行尾时才会发出这个宏编辑命令”。

除了上面这些条条框框，我们还希望大家记住这样一件事：既然打算定义一个宏，那么肯定是想反复多次地执行它。也就是说，几乎百分之百地会用到“**C-u C-x e**”（连续执行 4 次）、“**C-u C-u C-x e**”（连续执行 16 次），甚至“**C-u C-u C-u C-u C-u C-x e**”（连续执行 1024 次）等命令。只要稍有预见性，就能创建出可以连续执行无数次而不会出现问题的宏。

一般说来，好的宏是由 3 个部分组成的：

1. 找到准备开始工作的地方（通常利用查找操作来实现）。
2. 对文本进行预定的编辑处理。
3. 做好下次循环的准备工作。

那么，宏是如何做好下次循环的准备工作呢？请看下面这个例子。假设写一个用来删除某表格第三栏的宏。在删除了那一栏之后，这个宏应该位于下一行的行首（或者应该处于的其他地方）。这样，如果需要重复使用这个宏，就不用再去调整位置。

再来看一个稍微复杂点的例子。如果在某个宏的开头使用了一个查找操作，就必须在这个宏结束时把光标移动到本次查找点的后面。否则，宏将在文件的这个查找点

处陷入死循环，永远也到不了下一个查找点。作为一条基本规则，如果宏是用来对一个文本行进行操作的，那么它就应该以移动到下一行的行首作为结束。记住：目标是设计出一套能够连续地重复执行多次的击键动作，而整个过程应该是不会中断的。

体面地结束是设计宏编辑命令时应该努力追求的另外一个目标，而这往往又与选择录制的击键动作有关。比如，有几个 Emacs 命令（比如“**C-n**”）即使到达了编辑缓冲区的末尾也不会停止执行和报警，它会在文件尾添加新行。如果宏里有一个放错了位置的“**C-n**”而且还让它执行了好几百次，那么它即使到了应该停止的时候也不会停止。因此，如果想移动到下一个文本行的行首，就应该使用“**C-e C-f**”而不是“**C-a C-n**”。这两者的执行效果是一样的，但前者不会让文件无端地增加出空白行来。

示例：一个更复杂的宏

有时候，需要把文件里出现某个特定单词（或短语、句子等）的地方都找出来。下面这个宏能够把编辑缓冲区里所有包含单词“Emacs”的句子都找出来，并把它们放到另外一个编辑缓冲区里。如果想试试这个宏，需要先把一些关于 Emacs 的文字输入到某个编辑缓冲区里（如果手里没有现成的材料，可以照抄下面示例画面里的内容）。

键盘操作	动作
C-x (开始录制一个宏；状态行上将出现“Def”字样
C-s emacs	查找单词“Emacs”
RETURN	如果查找成功，停止查找操作；如果查找不成功，它将报警并停止这个宏的执行
ESC a	移动到句首“
C-@	设置文本块标记
ESC e	移动到句尾
ESC w	把句子复制到删除环里
C-x b emacsrefs RETURN	移动到名为“emacsrefs”的编辑缓冲区
C-y	插入这个句子
RETURN	为下一个句子准备一个新行

键盘操作	动作
C-x b RETURN	返回刚才的编辑缓冲区
C-x)	结束宏的录制工作；状态行上的“Def”字样将消失

- a. “**ESC a**”命令关于“句子”的概念由变量 **sentence-end** 控制。这个变量是一个相当复杂的正则表达式。在默认的情况下，句子将以句号(.)、问号(?)或惊叹号(!)结束；它们后面还可以再跟有引号或括号(包括方括号和花括号)；最后是两个以上的空格或一个换行符。

现在，假设已经录制好这个宏，可以用“**C-x e**”来执行它了。下面这个画面就是运行这个宏时所发生的情况。

按下： **C-u C-x e**

```
Buffers Files Tools Edit Search Help
Some Sentences About Editors

TECO is an archaic editor. Emacs is a modeless editor. vi is a modal
editor. EDT is a proprietary editor. GNU Emacs was written by
Stallman. Uniress Emacs was written by Gosling.
--- Emacs: emacstext (Text Fill)--L5--All-----
Emacs is a modeless editor.
GNU Emacs was written by
Stallman.
Uniress Emacs was written by Gosling.
--- Emacs: emacsrefs (Text Fill)--L5--All-----
```

通过执行这个宏，我们创建了一个名为“**emacsrefs**”的编辑缓冲区并把“**emacstext**”编辑缓冲区里出现单词“Emacs”的句子都复制到它里面来了。

完成操作时有可能看不到“**emacsrefs**”编辑缓冲区，但这不是故障：只要输入“**C-x b emacsrefs**”命令就能让这个新编辑缓冲区出现在屏幕上。

类似于上面这个例子中的情况，在定义一个宏时，可以在任意数量的编辑缓冲区之间来回切换。宏不必局限于一个编辑缓冲区。跨越多个编辑缓冲区的宏比较难于调试，涉及到的编辑缓冲区越多，要想记住光标和文本块标记的位置也就越难。另外，用户对正在访问哪一个编辑缓冲区这一问题的判断也更容易出现错误。因此，对于这种情况，明确地给出编辑缓冲区的名字（就像我们在例子里所做的那样）就将是一个比较好的做法。不过，一旦大家习惯了使用多个编辑缓冲区进行编辑，就肯定会对自己的工作之多感到惊讶。



窗口在某些宏里也很有用，同样必须给它们以足够的重视。比较好的做法是在屏幕上只有一个窗口的时候开始执行宏编辑命令，让宏来打开其他的窗口，到最后再（用“**C-x 1**”组合键）把窗口关闭得只剩下一个。如果在录制宏的时候屏幕上已有2个窗口，可在执行这个宏的时候屏幕上却有4个窗口，那么其后果将完全无法预料！一般来说，用“**C-x b buffername**”命令移动到一个有名字的编辑缓冲区要比用“**C-x o**”组合键移动到另一个窗口更值得信赖（后者会因过于含糊而不适用于一般情况）。这是因为所谓“另一个窗口”有可能是任何东西，比如“*Help*”编辑缓冲区、“*Completion*”编辑缓冲区、“*shell*”编辑缓冲区等等。移动到一个有名字的编辑缓冲区就精确得多，不管目标编辑缓冲区是如何（或者是否）显示在屏幕上的，它永远会进入正确的地方。

向现有的宏里增加编辑命令

虽然不能对宏进行编辑，但可以用“**C-u C-x (**”组合键在宏的尾部添加一些编辑命令。这个命令先执行完已经录制好的宏，然后会等待添加更多的击键动作。添加完击键动作之后，按下“**C-x)**”组合键将结束宏的录制工作。

这个命令特别适合用来补充不太完整的宏。如果宏编辑命令不太完整，解决方法之一是在宏执行完毕之后再把剩下的命令亲手输进去，另一种方法则是把必要的额外击键动作追加到这个宏的末尾。如果发现录制好的宏没有做好下次循环执行的准备工作，那么还是把必要的额外击键动作追加到这个宏的末尾比较好。在前面那个把包含单词“Emacs”的句子，复制到另一个编辑缓冲区里的例子里，我们最初录制的宏里就没有那个用来移动到第一个编辑缓冲区去的命令，它是后来才补充进去的。

示例

我们来看看下面这个示例。假设只想把文档各个段落里第一次出现单词“Emacs”的句子收集到一起，而这个文档恰好是在**troff**软件（它是UNIX内置的排版系统）里写成的。用**troff**写成的文档其各个节都是以“.**NH**”标记开始的，正好可以利用这一点识别出它们。我们来修改刚才的宏，把这个功能增加进去。如下所示：



键盘操作	动作
C-u C-x (开始录制一个宏；状态行上将出现“Def”字样
C-s .NH	已经找到单词“Emacs”并返回原来的编辑缓冲区。接下来查找下一节的开始
RETURN	如果查找成功，则停止查找操作
C-x)	结束宏的录制工作；状态行上的“Def”字样将消失

命名并保存宏

在很多时候，一个宏只是某个一次性问题的临时解决方法。它虽然能够减少工作量，但到不了保存起来以供今后再次使用的程度。可有时候，录制的宏能够完成需要经常做的事情，于是想把这些宏保存起来。比如，假设你是一位作家，经常需要把某个单词或者某个短语设置为某种字体，比如粗体等。虽然具体需要使用什么样的命令，取决于使用的是哪一种文本排版软件包，但录制一个宏来把单词设置为粗体肯定能简化操作。可是，不管录制宏的工作有多简单，每次录制这个宏时还是得输入一大堆的东西，你迟早会厌烦这种重复性的劳动。如果文字处理量确实很大，那么就可能需要在多次编辑会话中频繁地用到这个宏；于是，你既不想让它在你退出 Emacs 的时候消失，也不想让它在你录制另外一个宏的时候被覆盖。

说的再明确点，你想要的是一组能够随时用来把单词、句子和段落等设置成粗体字或斜体字的“永久性的”宏。如果你是一位程序员，那么需要的就是一些能够帮助对程序中的注释进行正确排版的宏（Emacs 已经在这方面为我们准备了一些内置的编辑功能，请参考第十二章）。

在这一小节里，我们将介绍怎样才能把宏保存起来，以便它们能够用在其他的编辑会话过程中。要想把一个宏保存起来，请按以下步骤进行操作：

1. 录制这个宏（如果已经录制好，就不用再做一次了）。
2. 输入“**ESC x name-last-kbd-macro**”并按下回车键。然后给宏起一个名字，再按下回车键。最好给它起一个不那么有“Emacs 味”的名字以避免让 Emacs 把宏和它自己的某个命令弄混。在进行了这样的操作之后，Emacs 就会记住这个宏，并且在以后的编辑会话里都不会忘记它。如果想使用这个宏，输入命令“**ESC x name**”（其中的“name”就是这个宏的名字）即可。

3. 如果想把这个宏定义永久地保存起来，还必须再进一步：必须把这个宏定义插入到一个文件里保存起来。用“**C-x C-f filename RETURN**”命令打开那个准备把宏定义插入到其中的文件，然后用“**ESC >**”组合键移动到它的文件尾。如果想让这个宏在每次进入Emacs的时候都可用，就需要打开“*.emacs*”文件；如果不是这样，则可以给那个文件起一个与宏相一致的名字，或者就用一个名为“*macros*”的文件把所有的宏定义都保存在里面。
4. 输入“**ESC x insert-kbd-macro RETURN macroname RETURN**”命令。Emacs 将把代表这个宏的 LISP 代码插入到这个文件里。能不能看懂这些代码并没有多大关系。
5. 按下“**C-x C-s**”组合键以保存那个文件。

现在，宏就永久性地保存在那个文件里了。

如果各位胆子够大，还可以亲自编辑那个文件以修改宏定义。但我们并不鼓励这样做。Emacs 保存起来的宏定义是击键动作的一个脚本，想编辑它是不容易的，稍有不慎，就会把事情搞成一团糟。

执行一个已命名的宏

给宏起了名字之后，就可以用“**ESC x macroname RETURN**”命令来执行它。还可以用“**ESC n ESC x macroname RETURN**”命令来重复执行这个宏“n”次。可以同时使用任意个有名字的宏而不仅仅局限于一个——就像键盘宏命令（keyboard macro）的情况一样。

如果把宏保存在“*.emacs*”文件里，要想执行它们就很容易：因为只要运行 Emacs，那些宏定义就会被加载进来。只需输入“**ESC x macroname RETURN**”命令即可。

如果把宏保存在其他的文件里，它们就不会被自动加载进来。在执行某个宏编辑命令之前，必须用“**ESC x load-file**”命令让 Emacs 读入宏定义文件。比如，假设定义了一个名为“*bold-word*”的宏，并且已经把它保存在子目录“*~/emacs*”（主目录中的 *emacs* 子目录）里的文件“*nroff.macs*”里。如果想使用这个宏，就先得执行“**ESC x load-file “~/emacs/nroff.macs RETURN**”命令。如果想让这个文件被自动加载进来，就必须把下面这条语句添加到“*.emacs*”文件里：

```
(load-file "~/emacs/nroff.macs")
```

不管宏定义文件是由用户本人手动加载的还是通过“*.emacs*”文件自动加载的，只要加载了这个文件，就随时都能发出“**ESC x bold-word RETURN**”命令。

示例

下面是个很有用的小技巧。假设已经录制并保存了一个名为“*make-double-spaced-text*”的宏。不过，要是每次用到这个宏的时候都得输入“**ESC x make-double-spaced-text RETURN**”就未免有点浪费时间了；而且，说实话，这还不如手动按回车键简单。那像这样的问题又该如何解决呢？可以定义一个键盘宏命令（keyboard macro）来执行永久性宏编辑命令。

键盘操作	动作
C-x (开始键盘宏命令的定义
ESC x make-double-spaced-text RETURN	执行保存起来的宏
C-x)	结束键盘宏命令的定义

现在，只需按下“**C-x e**”组合键，就能执行 **make-double-spaced-text** 宏。

再引申一步，还可以把宏编辑命令绑定到某个按键上。对这个问题的探讨请参考第十一章。我们先在这儿举一个例子，如果想把宏“*make-double-spaced-text*”绑定到“**C-c C-d**”组合键上，需要使用“**ESC x local-set-key C-c C-d make-double-spaced-text RETURN**”命令。（这种把宏绑定到某个按键上去的方法只对本次 Emacs 会话有效。如果想了解怎样才能让按键绑定成为永久性的，请参考第十一章内容。）

建立复杂的宏

我们已经把录制、执行和保存键盘宏命令的基本操作都介绍给大家了。下面再介绍几个 Emacs 允许用在宏编辑命令里的高级操作：暂停宏的执行以等待键盘输入和在宏里插入一个查询。

暂停宏的执行以等待键盘输入

有时候，暂停一个宏的执行以便通过键盘输入一些东西是很有必要的。比如，如果你有很多备忘录要写，就可以录制这样一个宏：它负责给出一个备忘录模板，而且会在适当时刻暂停执行以便填写有关的内容（比如日期和收信人等）。你可以用在宏里插入一个递归编辑的方法来完成这项工作（或者其他类似的工作）。这个“递归编辑”就好像是在说“先暂停一下，让我输入点东西，等我弄好了再继续执行宏编辑命令”。

在对宏进行定义的过程中，如果在某个地方按下了“**C-u C-x q**”组合键，就表示想在那里插入一个递归编辑；Emacs 也就会从那个地方开始进入一个递归编辑（状态行上将出现一对方括号，用户可以根据这一点来判断自己是否在递归编辑里）。在递归编辑期间输入的任何东西都不会被录制到这个宏里。可以输入任何东西，但结束时必须用“**ESC C-c**”组合键退出递归编辑。按下“**ESC C-c**”组合键时，状态行上的方括号将消失；而一旦画面上不再有方括号，就表示已经离开递归编辑——此后输入的任何东西又将被录制到宏里。可以根据需要在宏里插入任意多次暂停。

示例

请看下面这个示例。这个宏将在屏幕上显示一个备忘录模板，并且会用递归编辑等待输入日期、收信人姓名和发信人姓名。

键盘操作	动作
C-x (开始录制一个宏；状态行上将出现“Def”字样
Memorandum	把“Memorandum”显示到屏幕上
ESC s	使单词“Memorandum”居中
RETURN RETURN	把光标下移两行
Date: TAB	把“Date:”显示到屏幕上
C-u C-x q	进入一个递归编辑，在此期间中的击键动作不会被录制为宏的一部分
ESC C-c	退出递归编辑
RETURN RETURN	把光标下移两行
To: TAB	把“To:”显示到屏幕上
C-u C-x q	进入一个递归编辑
ESC C-c	退出递归编辑

键盘操作	动作
RETURN RETURN	把光标下移两行
From: TAB	把“From:”显示到屏幕上
C-u C-x q	进入一个递归编辑
ESC C-c	退出递归编辑
RETURN RETURN	把光标下移两行
ESC 78 -	用连字符画一条横穿屏幕的直线
RETURN	插入一个空白行
C-x)	结束宏的录制工作；状态行上的“Def”字样将消失

以下几个画面演示了这个宏的执行情况。

按下: **C-x e**

```
Buffers Files Tools Edit Search Help
Memorandum
Date: [REDACTED]
--***-Emacs: memo [(Text Fill)]--L3--All-----
```

宏暂停执行以等待输入日期。

输入: **10/31/97 ESC C-c**

```
Buffers Files Tools Edit Search Help
Memorandum
Date: 10/31/97
To: [REDACTED]
--***-Emacs: memo [(Text Fill)]--L5--All-----
```

宏暂停执行以等待输入收信人姓名。



输入: **Fred** ESC C-c

```
Buffers Files Tools Edit Search Help
                                         Memorandum

Date: 10/31/97

To: Fred

From: ■

--- Emacs: memo [(Text Fill)]--L7--All-----
```

宏暂停执行以等待输入发信人姓名。

输入: **Maurice** ESC C-c

```
Buffers Files Tools Edit Search Help
                                         Memorandum

Date: 10/31/97

To: Fred

From: Maurice

■

--- Emacs: memo [(Text Fill)]--L10--All-----
```

宏用短划线字符画出一条横贯画面的直线, 以隔开备忘录的标题部分和内容部分。

由宏负责的工作到这里就结束了; 接下来将由你来输入备忘录的内容部分。当然, 你也可以回头去编辑自己刚才填写的任何一组文字。

在宏里插入一个查询

对一个宏来说, 如果用它来完成的工作越复杂, 它的用途就越单一, 它的适用面也就越窄。虽然宏能够用来做很多的事情, 但它们毕竟不是程序——不能在宏里使用 if 语句、循环语句或者只有程序才具备的其他东西。说的再明确点, 宏不能从用户那里接受输入, 并根据这个输入采取必要的行动。如果各位需要的是这方面的功能, 请去阅读第十三章。

但我们还是有办法让宏从用户那里获得输入数据的, 虽然这个办法有一定的局限性。宏可以在运行时向用户提出查询; 它的工作情况与查询 - 替换操作很相像。如果想

录制一个这样的宏，就必须在这个宏的录制过程到达想让它暂停执行的地方时按下“**C-x q**”组合键。不会立即发生什么事情，可以像平时那样继续录制这个宏直到它完成。

可等到执行这个宏的时候，就会发生一些有趣的事情。当宏执行到按下“**C-x q**”组合键的地方时，Emacs 会在辅助输入区里给出一个如下所示的查询：

```
Proceed with macro? (y, n, RET, C-l, C-r)
```

下面是对这个查询的回答，它与我们在第三章里介绍的查询 - 替换操作的情况很相似：

- 回答“**y**”表示继续执行这个宏。执行完毕则进入下一次循环——如果有下一次循环。
- 回答“**n**”表示停止执行这个宏，但进入下一次循环（如果有）。
- 按回车键表示停止执行这个宏并取消尚未执行的循环。
- 按“**C-r**”组合键将开始一次递归编辑，可以做任何编辑或移动；退出这次递归编辑时，宏将从刚才被打断的位置开始准备继续执行。按“**ESC C-c**”组合键退出递归编辑；Emacs 会询问是否想继续执行这个宏，输入“**y**”表示“是”，输入“**n**”或回车键表示“否”。
- 按“**C-l**”组合键将把光标所在的文本行放到画面的中央（以便人们能够清楚地看到它前后的上下文）。类似于按下“**C-r**”组合键的情况，Emacs 也会询问是否想继续执行这个宏，输入“**y**”、“**n**”或回车键作为回答。
- 按“**C-g**”组合键（这个组合键没有被列为一个操作选项）取消这个查询和整个宏；它的作用类似于按下回车键。

示例

假设已经录制了一个用来把某程序的注释复制到另外一个编辑缓冲区里的宏。程序中的注释都被上、下两行星号包围着，所以这个宏将以查找一个星号字符串的查找命令开始。但并非所有的星号字符串后面都跟有注释，也并非所有的注释都值得复制。因此，在查找命令的后面加上一个查询，以便能够有选择地决定被找到的注释有哪些是值得复制的。

下面是这个宏的录制过程。

键盘操作	动作
C-x (开始录制一个宏；状态行上将出现“Def”字样
C-s *****	查找一个星号字符串
RETURN	如果查找成功，退出查找操作
C-x q	在宏里插入一个查询；Emacs会在宏执行到这个位置时询问是否想继续前进
C-e	移动到行尾
C-f	移动到下一行的行首
C-@	设置文本块标记
C-s ***** RETURN	查找配对的星号字符串
C-p	上移一行
C-e	移动到行尾
ESC w	把注释复制到删除环里
C-e C-f C-e C-f	把光标移动到这段注释的紧后面
C-x b comments	移动到名为“comments”的编辑缓冲区
C-y	把注释插入到编辑缓冲区里
RETURN	移动到下一行
C-x b	返回原来的编辑缓冲区
C-x)	结束宏的录制工作；状态行上的“Def”字样将消失

LISP 函数——宏的补充

宏是简化重复性工作的一种重要工具。它们允许各位用自己录制的命令来完成复杂的任务，而不需要去学习任何还不知道的内容，只要知道如何用基本的 Emacs 命令移动光标和编辑文本就足够了。即使是一位 Emacs 的初学者，也能毫无困难地使用宏。

Emacs 的灵活性几乎是无限的，但宏并不能解决所有的问题。在许多情况下，除了编写一个 LISP 函数来完成预期的工作之外没有任何其他的选择。如果你了解 LISP 或者愿意去学习这种程序设计语言，就可以自行编写 LISP 函数来解决那些比键盘宏

命令所能对付的情况更复杂的问题。我们将在第十三章介绍编写LISP函数的基本方法。

表 10-1 对这一章里介绍的与宏有关的编辑命令进行了汇总。

表 10-1：与宏有关的 Emacs 命令

键盘操作	命令名称	动作
C-x (start-kbd-macro	开始录制一个宏
C-x)	end-kbd-macro	结束宏的录制工作
C-x e	call-last-kbd-macro	执行最后一次录制的宏
ESC n C-x e	digit-argument 和 call-last-kbd-macro	把最后一次录制的宏执行n次
C-u C-x (universal-argument; start-kbd-macro	执行最后一次录制的宏,然后等待追加新的按键动作
(无)	name-last-kbd-macro	给刚才录制的宏起个名字(做好保存这个宏的准备)
(无)	insert-last-keyboard-macro	把起了名字的宏插入到一个文件里
(无)	load-file	加载内容为宏命令定义的文件(用来加载保存起来的宏文件)
(无)	宏名	执行保存起来的某个键盘宏命令
C-x q	kbd-macro-query	在宏定义里插入一个查询
C-u C-x q	(无)	在宏定义里插入一个递归编辑
ESC C-c	exit-recursive-edit	退出递归编辑



第十一章

对 Emacs 进行定制

本章内容：

- 键盘的定制
- 终端支持
- Emacs 变量
- Emacs 的 LISP 程序包
- 自动模式的定制

Emacs 几乎能按照所能想像到的任何方式进行定制。屏幕上显示的任何东西、任何命令、任何按键动作和任意一条消息等等几乎都能被定制。大家可能已经猜到了，几乎所有的定制工作都需要通过 Emacs 的启动文件 “`.emacs`” 来完成。

有些定制工作需要对 Emacs 中的 LISP 程序设计有一定的了解（请参考第十三章）；另外一些定制工作则相当简单，用不着那么大的学问。在这一章里，我们将向大家介绍一些简单而又不需要程序设计知识的定制工作。就目前情况而言，大家只要知道每一个 Emacs 命令都对应一个 LISP 函数就足够了。LISP 函数的基本格式如下所示：

(function-name arguments)

比如，如果想把光标往前移动一个单词，就需要按下“**ESC f**”组合键。而这个动作其实是通过调用下面这个 LISP 函数来实现的：

(forward-word 1)

这一章会介绍很多与 Emacs 的定制和它的高级用法有关的问题。我们将从读者最想知道，或者说最需要知道的两个与 Emacs 密切相关的事物（即键盘和终端）开始讲起。大家将学习怎样对键盘命令进行定制，怎样设置和使用键盘上的特殊键，怎样对终端进行设置才能让它正确地与 Emacs 配合工作。此外，本章还对 Emacs 变量和给 Emacs 添加额外功能的程序包进行了介绍，大家可以利用这些东西去改变和拓展

Emacs的操作行为。详细的变量清单和程序包清单收录在本书的附录三和附录四里。更高级的定制技巧请参考第十三章。

在我们开始探讨这些话题之前，有两件关于“`.emacs`”文件的重要事项要告诉给大家。首先，如果向“`.emacs`”文件插入代码时出了错，那么插入的东西就可能会造成 Emacs 不工作或者工作不正常。如果遇到这样的问题，有个简单的方法可以让 Emacs 在不运行“`.emacs`”文件的情况下启动：用命令行选项“`-q`”启动 Emacs，Emacs 将不运行“`.emacs`”文件。然后可以检查这个文件，看它在什么地方出了错。

其次是在定制 Emacs 环境方面我们所能给大家的最重要的一句忠告：多与其他用户进行交流。在终端配置问题上陷入困境或者在设置某个特殊编辑模式遇到麻烦的时候，其他用户可能早就遇到过并解决了这类问题。向他人求助决不是“不诚实”或者“懒惰”，事实上，Emacs 的缔造者是鼓励这样做的、他们更愿意看到他们的软件被大家所共享而不是独藏。Emacs 甚至提供了一种测试其他用户的“`.emacs`”文件的简单方法：用选项“`-u username`”启动 Emacs，则该 `username` 的“`.emacs`”文件将取代原先的“`.emacs`”文件运行。

键盘的定制

也许 Emacs 用户最想定制的东西就是运行 Emacs 编辑命令的按键动作了。我们说过，按键动作是通过“按键绑定”与命令关联起来的。

事实上，Emacs 里的每个按键动作都会运行一个命令。对应于可打印字符（字母、数字、标点符号和空格）的按键所运行的命令是 **self-insert-command**，其作用是把被按下的键插入到当前编辑缓冲区的光标位置上。（可以给 Emacs 新手开个愚人节玩笑——把他们的可打印字符的按键绑定改掉。）

对大多数操作而言，默认的按键绑定集是很适当的；但在很多情况下，可能需要添加或者改变按键绑定。Emacs 有好几百条命令，其中只有一部分带按键绑定。正如大家已经知道的，输入“**ESC x command-name RETURN**”就能输入那些不带按键绑定的命令。

如果经常要用到一个不带绑定的命令，就可以考虑把它绑定到一个按键组合上以方

便工作。有的人也许想把键盘上的特殊键（比如箭头键、数字小键盘或者功能键等）与经常使用的命令关联起来。如果你使用的是一个运行X窗口系统的图形终端，那么可能还需要对鼠标动作和菜单选项进行定义。这一章将介绍一些按键绑定和修改按键绑定的例子，鼠标和菜单事件的绑定将留做第十四章的内容。

另一个现在就必须弄清楚的重要概念是“键位映射图”(keymap)，它是多个按键绑定构成的一个集合。Emacs最基本的默认按键绑定保存在一个名为“**global-map**”的全局键位映射图里。与全局键位映射图相对应的是“局部(local)键位映射图”，每个编辑缓冲区都有一个自己的局部键位映射图。局部键位映射图的作用是实现某具体编辑模式（比如C模式、文本模式、shell模式等）里的编辑命令，不同的编辑模式有它们各自的键位映射图；进入某个编辑模式时，它所对应的键位映射图就会被安装为局部键位映射图。按下某个按键的时候，Emacs会先查看当前编辑缓冲区的局部键位映射图（如果有）里有没有对它进行定义。如果Emacs没有在局部键位映射图里找到与按键对应的定义项，就会查看**global-map**键位映射图。如果找到了与按键对应的定义项，那么按键的关联命令就会被执行。

那么，绑定到组合键上的命令，比如“**ESC d**”（命令名是**kill-word**），是怎样被查到和执行的呢？情况是这样的：“**C-x**”、“**ESC**”和“**C-c**”等按键其实都被绑定到一些特殊的内部函数上，而这些函数会让Emacs等到另一个按键被按下后再去另外一个键位映射图里查找该按键的绑定定义；如果后面的按键在一秒钟后还没有被按下，那么会在辅助输入区里显示“**C-x-**”。供“**C-x**”和“**ESC**”使用的键位映射图分别叫做**ctl-x-map**和**esc-map**；而“**C-c**”组合键则是为关联于某具体编辑模式（比如C模式和shell模式等）的局部键位映射图保留的。

举个例子，按下“**ESC d**”组合键的时候，Emacs将先到该编辑缓冲区的局部键位映射图里去查找，假设它没有在那里找到与之对应的定义项。于是，Emacs会再到**global-map**里去查找。Emacs在这里找到一个对应于“**ESC**”的定义项，它定义了一个特殊的函数（叫做“**ESC prefix**”）；这个函数将等待下一个按键的到来，然后根据**esc-map**键位映射图来决定要执行哪一个命令。现在，按下“**d**”的时候，“**ESC prefix**”函数将在**esc-map**键位映射图里查找有无对应于字母“**d**”的定义项，它找到了**kill-word**命令，于是就执行这个命令。

通过在键位映射图里增加（或者改写现有的）定义项的方法，能创造出自己的按键

绑定。有 3 个函数可以用来完成这一操作，它们是 **define-key**、**global-set-key** 和 **local-set-key**。这 3 个函数的格式如下所示：

```
(define-key keymap "keystroke" 'command-name)
(global-set-key "keystroke" 'command-name)
(local-set-key "keystroke" 'command-name)
```

注意，“*keystroke*”要用双引号引起来，“*command-name*”的前面要用单引号。这是 LISP 语言的语法所要求的，详细情况请参考第十三章。“*keystroke*”是一个或者多个字符，可打印字符或特殊字符都可以。如果是特殊字符，就必须使用表 11-1 给出的表达记号。

表 11-1：特殊字符的表达记号

特殊字符表达记号	定义情况
\C-x	C-x (其中“x”是任意字符)
\C-[或 \e	ESC；转义符
\C-j 或 \n	LINEFEED；换行符
\C-m 或 \r	RETURN；回车符
\C-i 或 \t	TAB；制表符

于是，字符串 “abc\C-a\ndef” 就等于是把 “abc”、“C-a”、换行符 LINEFEED 和 “def” 合并起来而得到的一个字符串。注意，控制字符是不区分大小写的。换句话说，“\C-A” 和 “\C-a” 是一样的。

函数 **define-key** 是最通用的，它可以用来自定义任何一个键位映射图里的按键。**global-set-key** 只能绑定全局键位映射图里的按键，因为 **global-map** 只有一个；“(**global-set-key** ...)”和“(**define-key global-map** ...)”的作用完全一样。**local-set-key** 的作用是绑定当前编辑缓冲区局部键位映射图里的按键，因此它只能用来在某次 Emacs 会话期间临时性地定义按键绑定。

下面是一个简单的键盘定制示例。假设你正用某种程序设计语言写程序代码。对程序进行编译时，编译器给出的出错信息带有出错语句的行号，想快速移动到源代码文件的那一行去改正错误（注 1）。这需要使用 **goto-line** 命令，但这个命令在默认的

注 1：对付这种情况其实有一个更好的办法，我们将在下一章对此做详细讨论。

情况下没有被绑定到任何按键上。现在，假设想把它绑定到“**C-x l**”组合键上。于是，需要添加到“*.emacs*”文件里的语句应该是：

```
(define-key ctl-x-map "l" 'goto-line)
```

这就把**ctl-x-map**键位映射图里的“l”字母格与**goto-line**命令绑定在一起。也可以用下面两条语句中的任何一条来达到同样的目的：

```
(define-key global-map "\C-xl" 'goto-line)
(global-set-key "\C-xl" 'goto-line)
```

这两条命令能够达到同样的目的，而且使用起来可能更方便，因为很可能记不住对应于“**C-x**”的键位映射图叫做**ctl-x-map**。

再看一个例子。我们在第二章里把“**C-x ?**”组合键绑定为**help-command**命令，把“**C-h**”组合键绑定为**backward-char**命令。这两个按键绑定的定义情况如下所示：

```
(global-set-key "\C-x?" 'help-command)
(global-set-key "\C-h" 'backward-char)
```

这两个定义也可以写成下面这样：

```
(define-key ctl-x-map "?" 'help-command)
(define-key global-map "\C-h" 'backward-char)
```

把一个按键绑定插入到“*.emacs*”文件里之后，需要“运行”（或者叫“求值”）这个文件以便改动能够生效。用来完成这一工作的命令是“**ESC x eval-current-buffer RETURN**”。更好的方法是按下“**C-x C-e**”组合键，它（我们将在下一章对它做进一步的讨论）的作用是只执行光标所在的那一行上的LISP代码。如果没有进行上述操作，那么所做的改变就只有等到下次启动 Emacs 时才会生效。

特殊键

更复杂的键盘定制任务是把命令绑定到终端上的特殊键上，比如方向键、数字小键盘或者功能键等。这个级别的定制需要多做一些准备工作。如果各位喜欢使用特殊键，那么多花些功夫还是值得的。

不论什么型号的终端，它上面的特殊键都会产生一组“字符代码”，即由多个字符组

成的一个序列，这些序列通常由一个特殊的字符（比如“ESC”字符）打头。因此，需要做的第一件事是找出终端上的特殊键都会产生哪些字符代码。比如，许多终端都遵守 ANSI 标准，即当数字小键盘切换到应用模式（application mode，即 PC 键盘上 NumLock 指示灯不亮时的状态）时，它们能够产生复杂的代码序列；而切换到普通的数字模式（即 PC 键盘上 NumLock 指示灯亮时的状态）时，数字小键盘上的按键输入的将是数字和标点符号（英文句号、逗号和短划线）。在应用模式下，数字小键盘所产生的字符代码全都以“**ESC O**”打头，后面跟着一些其他的字符，比如，按键“7”所对应的字符代码将是“**ESC O w**”。

终端上的特殊键所产生的字符代码，可以在该终端的使用手册里查到。如果找不到有关资料，可以用 **quoted-insert** 命令把那些字符代码查出来。具体操作是：先切换到一个空白的编辑缓冲区里——比如“*scratch*”编辑缓冲区，然后先按下“**C-q**”组合键（注 2），再依次按下各特殊键。控制字符的前面会有一个“^”字符，比如“**^X**”代表的就是“**C-x**”（注 3）。有些特殊字符也会被显示为控制字符，比如下面这几个：

ESC 转义符	^[
RETURN 回车符	^M
LINEFEED 换行符	^J

TAB 制表符将以制表位（即一个或者多个空格）的面目出现。所以，数字小键盘上的按键“7”所对应的字符代码将是“**^[Ow**”。

但这种利用“**C-q**”组合键的技巧有很大的局限性，它只能对单个字符进行处理。因此，如果特殊键会产生多于一个字符的序列，则从第二个字符开始，以后的字符都必须是普通的（可打印的）；否则，Emacs 就将把它们解释为编辑命令。但根据有关标准，任何键盘上的特殊键所产生的字符代码其构成都是一个特殊字符（比如转义符“**ESC**”）加几个可打印字符，所以从原则上讲应该是安全的。

某些终端上的特殊键不适合做 Emacs 定制，因为它们产生的字符代码与某些基本的 Emacs 命令一模一样。比如，Wyse WY-50 终端上的方向键所产生的字符代码就分

注 2： 或者“**ESC q**”组合键——如果遇到前面介绍过的流控制问题。

注 3： 注意：控制字符虽然被显示为两个字符，可它们实际占用的只是一个字符的位置。如果在光标位于某个控制字符上时按下“**C-f**”组合键，则光标将向右移两个字符。

别是“**C-k**”（上）、“**C-j**”（下）、“**C-h**”（左）和“**C-l**”（右）；完全没有必要去重新绑定这些按键！有些终端虽然有数字小键盘，可它却没有应用模式。换句话说，它们的数字小键盘只能产生数字和标点符号。很明显，这样的数字小键盘也不能用于Emacs定制。这里有一条基本规则：只有在特殊键所产生的字符代码是以一个特殊字符（比如转义符“**ESC**”）开头，并且至少有3个字符长时，才能在Emacs里对它们进行定制。

如果终端上的特殊键符合这个要求，那么可以在“*.emacs*”文件里写一些LISP代码把这些特殊键和挑选出来的命令绑定在一起。这些LISP代码必须先创建一个与特殊按键的常用前缀相对应的键位映射图，然后再把各个特殊键绑定到某个命令上。请看下面这段LISP代码，它可以用来绑定ANSI兼容终端（比如DEC VT100或VT200型终端）的方向键、功能键和数字小键盘上的按键：

```
(setq term-file-prefix nil)
(send-string-to-terminal "\e=")

(setq SS3-map (make-keymap)) ;set up the keymap for the keypad
(define-key global-map "\eO" SS3-map)

(define-key SS3-map "A" 'previous-line) ; up arrow
(define-key SS3-map "B" 'next-line) ; down-arrow
(define-key SS3-map "C" 'forward-char) ; right-arrow
(define-key SS3-map "D" 'backward-char) ; left-arrow

(define-key SS3-map "M" 'isearch-forward) ; Enter

(define-key SS3-map "P" 'other-window) ; PF1
(define-key SS3-map "Q" 'query-replace) ; PF2
(define-key SS3-map "R" 'rreplace-string) ; PF3
(define-key SS3-map "S" 'suspend-emacs) ; PF4
(define-key SS3-map "L" 'delete-char) ; ,

(define-key SS3-map "n" 'delete-next-word) ; .
(define-key SS3-map "p" 'isearch-backward) ; 0
(define-key SS3-map "q" 'beginning-of-line) ; 1
(define-key SS3-map "r" 'find-file) ; 2
(define-key SS3-map "s" 'end-of-line) ; 3
(define-key SS3-map "t" 'backward-word) ; 4
(define-key SS3-map "v" 'forward-word) ; 6
(define-key SS3-map "w" 'beginning-of-buffer) ; 7
(define-key SS3-map "x" 'save-buffer) ; 8
(define-key SS3-map "y" 'end-of-buffer) ; 9
```

在上面的代码里，分号后面的文字是一些注释，Emacs不会执行它们。

这段代码的头两行需要做些解释。Emacs有一些内建的LISP代码能够对某些型号的终端（尤其是DEC VT100和VT200系列以及各种工作站）上的特殊键进行绑定，但对其他型号的终端就无能为力了。上面这段代码的第一行语句将消除Emacs内置LISP代码（如果有的话）的设置情况，从而确保这段代码的其他语句对任何型号的终端都能正确工作。

第二行语句是专门为VT100 / VT200系列终端而准备的，这类终端的数字小键盘在应用模式和数字模式下都能使用。在数字模式下，数字小键盘上的按键只是数字、短划线、逗号和英文句号的“副本”。很明显，这个模式在我们的示例里是没有用的；所以我们需要强制终端进入应用模式。向终端发送“**ESC =**”序列就能达到这一目的，这就是第二行中**send-string-to-terminal**命令的由来。注意：如果终端不是VT100、VT200或兼容型号，就不要使用这一行。

再看接下来的两行语句。它们一个用来创建键位映射图，一个用来把新键位映射图挂到全局键位映射图上，好让新键位映射图发挥作用。第一条语句创建了一个名为**SS3-map**的键位映射图；可以随便给自己的键位映射图起名字，我们之所以会给这个键位映射图起一个这样的名字是因为DEC终端的文档把“**ESC O**”前缀叫做“SS3”。第二条语句把这个键位映射图绑定到全局键位映射图的“**ESC O**”序列上（注4）。请注意：不仅仅是编辑命令，键位映射图也可以被（递归地）绑定到键位映射图里的字母格里。于是，只要“**ESC O**”被键入，即只要某个能产生一个以“**ESC O**”打头的字符序列的按键被按下，这个LISP命令就会让**global-map**键位映射图把控制权传递给**SS3-map**。

上面这段代码中的其他LISP命令把各方向键和数字小键盘上的按键分别绑定到相应的Emacs编辑命令上。4个方向键被分别绑定到相应的光标移动命令上，而数字小键盘上其他按键的绑定情况如下图所示：

注4： 它实际上是指**SS3-map**绑定到**esc-map**的“O”字母格里；有关说明请参考本章开头部分对键位映射图的解释。

F1 移动到 其他窗口	F2 查询 - 替换	F3 替换 字符串	F4 挂起 Emacs
7 移动到编辑 缓冲区的开始	8 保存编辑 缓冲区	9 移动到编辑 缓冲区的末尾	-
4 后移一个 单词	5	6 前移一个 单词	， 删除一个 字符
1 移动到 行首	2 查找文件	3 移动到 行尾	ENTER 向文件头方向 开始递增 查找工作
0 向文件尾方向 开始递增查找工作		- 删除一个 单词	

注意：按键“5”和“-”（短划线）没有被绑定。作为一个练习，想想需要写出什么样的 LISP 代码才能把这两个按键绑定到自己最常用的两个 Emacs 编辑命令上。

终端支持

显示设备的种类有很多，比如终端、PC 个人电脑和图形工作站等等。只要不是“最笨的”显示设备，Emacs 就都能在其上运行；但它能否顺利工作则要取决于几个 UNIX 因素。

第一个因素叫做“环境变量”。登录到一个 UNIX 系统上的时候，这些变量中已经有几个是设置好的。环境变量的值（通常是一些字符串）由 UNIX 操作系统的 shell 负责管理，它们的作用是作为计算环境的各项特征的指示，shell 会把它们传递到发出的每一个命令上。

为了便于区分，环境变量的名字全部采用大写字母。典型的环境变量包括 **PATH** 和 **SHELL**，前者告诉 UNIX 要在哪些目录里查找命令，后者的值是当前使用的 shell 的名字。我们关心的环境变量是 **TERM**，它的作用是告诉 UNIX 操作系统和有关程序（比如 Emacs）正在使用哪一种终端。

TERM一般在你登录时设置，具体负责这项工作的或者是某个系统全局性的初始化文件，或者是你自己的 “*.profile*” 文件（C shell 用户使用的是 “*.login*” 文件）。在



某些系统上，甚至需要在登录时给出终端类型。要想查看 **TERM** 的值，请在 UNIX 提示符处输入 “**echo \$TERM**” 命令；美圆符号 “\$” 表示想查看的是环境变量的值。如果想对 **TERM** 进行设置，请输入 “**TERM=termname**”（适用于 Bourne、Korn 和 bash 等几种 shell。注意等号两端不能有空格）或 “**setenv TERM termname**”（适用于 C shell）命令。如果不知道自己的终端类型，可以询问系统管理员。如果他也不知道，请继续往下看，你很快就会知道怎样才能把它查出来。

与终端有关的另一个 UNIX 因素是目录 */etc* 里的 **termcap** 文件，这个文件对终端的物理指标进行了定义，比如字符的宽度和高度、终端本身用来完成卷屏、反显和光标移动等操作的硬件命令等。

从 AT&T System V 演化而来的 UNIX 版本使用的是 **terminfo** 而不是 **termcap** 文件；请询问系统管理员你所使用的 UNIX 属于哪一个系列。**terminfo** 使用了一个比单个文件更复杂的数据库来描述终端的物理指标，但保留了许多与 **termcap** 文件相接近的特点（基本上做到了向上兼容）。

termcap 数据项写得很精炼，非专业人员一般不容易看懂；而 **terminfo** 数据项是经编译得到的机读格式。不过，两种情况所支持的终端的名字还都比较容易辨认。

如果所用 Emacs 版本使用的是 **termcap**，那么就能用 UNIX 的 **grep** 工具来搜索 */etc/termcap* 文件。先根据它的制造商和型号猜测一个终端类型，然后输入命令：

```
* grep -i guess /etc/termcap
```

“**-i**” 选项的作用是不让 **grep** 区分大小写。假如终端的型号是 DEC VT100，就输入 “**grep vt100 /etc/termcap**” 命令。如果成功，**grep** 将显示一些下面这样的输出行：

```
vt100|vt100-am|dec vt100:\
```

用垂直线字符隔开的字符串都可以用做环境变量 **TERM** 的取值。这些字符串是同一种终端的不同名称，**termcap** 文件里的大多数条目都是这样的。

如果所用 Emacs 版本使用的是 **terminfo**，就可以通过 **terminfo** 数据库来检索终端类型。这个数据库组织结构为目录 */usr/lib/terminfo* 下的一个目录树。这个目录树由很多（下级）目录构成，这些下级目录的名字都只有一个字符，或者是一个数字，或者是字母表里的一个字母。终端类型按其名称的第一个字符（数字或字母）被分别

归类到相应的下级目录里，下级目录里的每个文件代表着一个终端类型。比如，文件 `/usr/lib/terminfo/v/vt100` 里保存的就是 VT100 的数据项。这些文件都是独立的终端描述文件。

有两种方法可以查出终端的 **terminfo** 名称。第一种方法是先根据所猜的终端名称的第一个字母用 **cd** 命令进入 `/usr/lib/terminfo` 子目录里相应的下级子目录里，再用 **ls** 命令列出其中的终端描述文件并把最有“嫌疑”的那个文件挑出来。用这个文件名作为终端类型，把环境变量 **TERM** 设置为它的名字。

检索 **terminfo** 数据库的另一个方法是使用 UNIX 操作系统的 **find** 命令。输入：

```
% find /usr/lib/terminfo -name "guess" -print
```

注意，“guess”必须放在双引号里，并且允许包含匹配文件用的通配符，比如“*”和“?”。如果检索成功，**find** 会把它找到的终端描述文件的名字列出来；再从中挑选一个作为环境变量 **TERM** 的值即可（不用加上目录名作为前缀）。

termcap 文件（和 **terminfo** 数据库）的内容随系统的不同而千差万别。在 UNIX 作为商业产品的早期，某 UNIX 系统的 **termcap** 文件不支持各种流行终端并不是件稀奇的事。随着 UNIX 操作系统的发展，**termcap** 文件也在成长；如今，一个 **termcap** 文件往往能支持好几百种终端类型。但尽管如此，有的终端还是有可能没有被包括在 **termcap** 或 **terminfo** 里；如果终端是由某个独立制造商（比如 WYSE）生产的，那么就更有可能如此。不过，这类终端通常是主流终端的克隆产品（仿制 VT100 的情况最常见）或者可以按主流终端进行配置。可以让自己的终端仿真某个主流的产品，然后再把环境变量 **TERM** 设置为相应的终端类型。一般说来，大多数 **termcap** 文件和 **terminfo** 数据库都完全不支持终端的情况是极为罕见的（注 5）。

许多功能——比如字符的插入功能和屏幕的部分卷屏功能——大多数现代终端用硬件就能实现，这就大大减轻了 Emacs 的负担。如果某项功能可以由终端的硬件来完成，Emacs 将坐享其成；如果某项功能不能由终端的硬件来完成，那么 Emacs 能用软件仿真出同样的效果。Emacs 甚至知道如何在缺少某些基本功能，比如反显功

注 5：对 **termcap** 文件和 **terminfo** 数据库的详细讨论请参考《termcap and terminfo》一书，它的作者是 John Strang、Linda Mui 和 Tim O'Reilly，由 O'Reilly & Associates 出版公司出版。

能 (Emacs 的状态行要用到这一功能) 的情况下“委曲求全”。只有一项功能是终端所必须具备的——它必须能显示光标。如果启动 Emacs 时看到一条消息告知终端功能不足，就表明或者是 **TERM** 环境变量设置不正确、或者是所用终端太“笨”；这第二种可能性如今已经是非常罕见了。

关于终端我们再多说几句。你可能会发现 **termcap** 或 **terminfo** 里有好几项看起来都适用于你的终端。对于这种情况 (假设使用的是 **termcap**)，排在第一个的那一项最有可能是正确的，或者至少是一个合理的默认值；**termcap** 文件通常就是按这个原则设计出来的。

有的终端在模仿某知名品牌 (比如 VT100) 的同时可能还具备某些额外的功能。对于这种情况，如果指定的是它实际的终端类型，而不是让它“自贬身份”去仿真最低档的主流品牌，那么 Emacs 也许能运行得更有效率。为了在 **termcap** 或 **terminfo** 里找到一个能够利用那些额外功能的设置项而多花些功夫是非常值得的。

流控制问题的解决办法

流控制 (flow-control) 问题指的是一种与终端支持有关的设备冲突现象，不少终端 (虽然如今已不多见) 就是因这个问题的影响才不能与 Emacs 很好地配合工作的。这个问题是这样的：为了预防缓冲区溢出，有些操作系统或数据通信设备 (终端、调制解调器、网络等等) 把 “C-s” 和 “C-q” 字符用做终端输入数据的“停止”和“前进”信号。

如今，使用其他流控制手段 (比如额外的硬件) 的数据通信系统越来越多，但仍有一些还在使用 “C-s / C-q” 方法。用户和作为最终目标的计算机之间的通信环节越多 (比如一个终端先连接到一个调制解调器，再连接到另外一个调制解调器，再连接到一个局域网，再连接到一台计算机，再远程登录到另外一台计算机……的情况)，出现这种问题的概率也就越大。

一般说来，用户和计算机之间的硬件设备越多，“C-s” 和 “C-q” 被用做流控制字符的可能性也就越大；而这将使 Emacs 无法在编辑命令里使用这两个字符。有个简单的方法可以查明这种情况是否会影响自己：启动 Emacs 并按下 “C-s” 组合键。如果能够在状态行上看见 “Isearch:” 字样，就说明一切正常；否则，看起来好像是什么事情也没有发生，可终端却可能已经被挂起来了，再按任何键，终端都没有反

应。实际情况是这样的：按下的“**C-s**”字符告诉某些硬件不要再继续接受输入数据。按“**C-q**”组合键退出这种局面，在这期间按下的那些键又都起作用了。

Emacs准备了一个简单的方法来解决这个问题。命令**enable-flow-control**的作用是让Emacs把“**C-s**”和“**C-q**”字符直接传递给底层的操作系统，操作系统将把它们看做是流控制命令并进行相应的处理；需要用“**C-**”（“control”键加反斜线键）和“**C-^**”（“control”键加“shift”键加“6”键）来分别代替原来的“**C-s**”和“**C-q**”组合键——不论可能会在什么地方用到它们。这样，对应于编辑命令**save-buffer**的按键绑定就将变为“**C-x C-**”。

还可以对Emacs做这样的设置：只要使用某种类型的终端，就自动激活流控制功能。这需要把下面这条语句添加到“*.emacs*”文件里：

```
(enable-flow-control-on termtypes)
```

语句中的“*termtypes*”是一个或者多个字符串，字符串里包含着想为之激活流控制功能的终端类型的**TERM**名称。请看下面这个示例。如果偶尔会在一台DEC VT-320终端上使用Emacs——这种终端上的流控制问题必须由它自己去处理，就要使用下面这条语句：

```
(enable-flow-control-on "vt320")
```

这里介绍的流控制命令已经在Emacs老版本的基础上有了很大的改进，在老版本里解决流控制问题还得再多做很多事呢。

Emacs 变量

下面再来说说如何改变Emacs的行为——而不仅仅是它的用户界面。达到这一目的最简单的方法是对控制着Emacs各种行为的变量进行设置。我们在第二章已经见过**auto-save-interval**等几个例子。要想设置Emacs变量，就需要在“*.emacs*”文件里使用**setq**函数，比如下面这样：

```
(setq auto-save-interval 800)
```

auto-save-interval变量取的是一个整数值；但很多Emacs变量取的是表示“真”或“假”的布尔值。在Emacs LISP里，真值用“**t**”来表示，假值用“**nil**”来表示；但



在很多情况下，任何不是“nil”的东西都可以用来代表真值。Emacs 变量还可以取其他类型的值，它们的表达方法如下：

- 字符串值——必须放在双引号里。我们在本章前面介绍按键绑定命令的参数时见过不少的字符串值。
- 字符值——类似于字符串，但字符值必须有一个问号（?）做前缀，而且不能放在双引号里。比如，“?x”和“?\C-c”就分别是“x”和“C-c”的字符值。
- 符号值——以单引号（'）打头，后面跟着一个符号名。比如像“'never”这样（参见附录三中的变量 **version-control**）。

附录三对常用的 Emacs 变量进行了归纳整理，分门别类地列出了它们的说明和默认值。Emacs 变量非常多——附录三里只列出了其中的一部分。可以这么说，只要想对 Emacs 的某个方面进行定制，就肯定会有个与之对应的控制变量（特别是当想改变的东西涉及到一个数字或者涉及到真、假条件时更是如此）。可以用将在第十六章介绍的 **apropos** 命令查出是否有与想定制的东西有关的变量，仔细看看它输出结果里的变量和它们的说明。

有些 Emacs 变量对不同的编辑缓冲区可以取不同的值（我们称这些值为局部值），与局部值相对的是默认值。如果没有给这些变量设定局部值，它们在编辑缓冲区里的取值就是其默认值。可以分别对这类变量的默认值和局部值做相应的设定。在设置 **left-margin** 或 **case-fold-search** 等变量的值的时候，实际设定的是它们的局部值。如果想设置这类变量的默认值，就必须使用 **setq-default** 命令而不是使用 **setq** 命令。如下所示：

```
(setq-default left-margin 4)
```

令人遗憾的是，没有什么通用的方法能够用来判断某个变量是只有一个全局性的取值，还是既有默认值又有局部值（除非阅读该编辑模式的 LISP 代码）。所以，我们认为最佳策略应该是先尽量使用 **setq** 命令，如果在实际工作中发现自己用 **setq** 命令设置给某个变量的值似乎没有起作用，那么再去使用 **setq-default** 命令。比如，假如已经把下面这条语句：

```
(setq case-fold-search nil)
```

添加到 “.emacs” 文件里，但发现 Emacs 的查找命令还是忽略字母的大小写（就好像这个变量的取值还是 “t” 一样），那么就应该再用 **setq-default** 命令设置一次。

Emacs 的 LISP 程序包

Emacs 包含大量的 LISP 代码。事实上，就像将在第十三章里看到的那样，Emacs 的内置功能基本上都是用 LISP 语言写成的。Emacs 还另外带有一些 LISP 程序包（或者叫程序库），可以随时用加载 LISP 程序包的方法来扩展 Emacs 的功能。为 Emacs 开发 LISP 程序包的工作一直在进行着，而且来源不仅限于自由软件基金会；有时候，系统管理员也会安装一些非自由软件基金会来源的程序包。

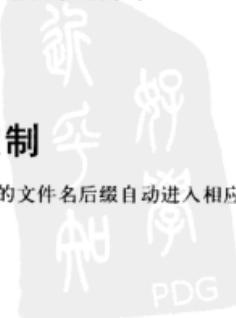
我们在附录四里用一个表格对比较常用的内置 LISP 程序包进行了汇总，对它们的使用方法也做了简单的介绍。如果想知道自己的系统上都有哪些程序包可用，请按下 “C-h p” 组合键（命令名是 **finder-by-keyword**）。简单地说，Emacs 内置的 LISP 程序包可以用来完成以下一些工作：

- 支持用 C、LISP、Perl、FORTRAN 及另外几种程序设计语言编写程序（参见第十二章）。
- 支持用 **nroff**、**troff**、**TeX**、**LATEX** 和 **HTML** 做文字排版处理（参见第九章）。
- 仿真其他编辑器（**vi**、**EDT** 和 Gosling Emacs 等）。
- 提供到其他 UNIX 实用程序的接口，比如 shell 和 mail（参见第五章和第六章），以及 FTP、Telnet、Kermit 和 MH 电子邮件系统等。
- 提供辅助编辑功能，比如拼写检查（参见第三章）、大纲编辑（参见第八章）、文本排序、编辑命令的历史记录和设置 Emacs 变量（参见附录三）等等。
- 提供各种游戏或者其他形式的娱乐等。

详细资料请参考附录四。

自动模式的定制

Emacs 能够根据准备打开的文件名后缀自动进入相应的主编辑模式，附录四的表格



里就列出了一些具备这种功能的主编辑模式。表格最右一栏给出了许多 Emacs 在默认情况下设定的文件名后缀与主编辑模式的关联关系。这些关联关系由专用的 Emacs 变量 **auto-mode-alist** 负责管理。这个变量是一个 “(*regexp . mode*)” 形式的数据组列表，其中 “*regexp*” 是一个正则表达式（参见第三章和第十三章），“*mode*” 是一个用来启动某个主编辑模式的函数的名字。Emacs 在打开文件的时候，会先到这个列表里（从头开始）查找有没有与文件名匹配的正则表达式。如果找到一个，就会运行与该正则表达式关联的函数以启动相应的主编辑模式。注意，文件名的任何一个部分（不仅仅是它的后缀名）都可以用来与某个主编辑模式建立关联关系。

也可以把自己的关联关系添加到 **auto-mode-alist** 列表里；不过，如果不熟悉 LISP，最好还是别这么做（参见第十三章）。假设正在用 Ada 语言写程序，而 Ada 编译器要求文件名后缀都是 “.a”（有些 Ada 编译器要求使用 “.ada”）。如果想让 Emacs 做到只要访问的是 Ada 文件，它就自动把它们放到 Ada 模式里，那么就需要把下面这条语句添加到 “*.emacs*” 文件里：

```
(setq auto-mode-alist (cons '("\\.a$" . ada-mode) auto-mode-alist))
```

请注意，在 “*cons*” 的后面必须有一个单引号 (')，在 “\\.*a\$*” 和 “*ada-mode*” 之间必须有一个英文句号 (.). 记号 “'(x . y)” 是用来“把 x 和 y 配对组成一个数据组”的 LISP 语法。字符串 “\\.*a\$*” 是个正则表达式，能够匹配“以 *a* 结尾的任何东西”；“\$”的意思是匹配字符串的结尾（注意不是匹配文本行的结尾——那是在正则表达式查找-替换操作中才需要匹配的东西）。整个 LISP 语句的意思是“把数据组 (\"\\.a\$\", 'ada-mode) 插入到 **auto-mode-alist** 列表的开始位置处”。注意，因为 Emacs 是从头开始检索 **auto-mode-alist** 变量的，并且会在找到一个匹配之后停止前进，所以可以用上面这种 “*cons*” 结构覆盖现有的编辑模式关联关系（注 6）。

再来看看另外一个例子。假设把某些电子邮件保存在文件名以 “msg-” 打头的文件里。如果想在文本模式里对这些文件进行编辑，就可以像下面这样做：

```
(setq auto-mode-alist (cons '("^msg-" . text-mode) auto-mode-alist))
```

注 6：LISP 程序员应该知道还有其他方法可以用来往 **auto-mode-alist** 列表里添加数据组，比如 “**append**（追加）”。

注意，在这个例子里，需要匹配的是文件名的开头而不是文件名的结尾。正则表达式操作符 (^) 表示匹配字符串的开始，所以整个正则表达式的意思是“以 msg- 开头的任何东西”。

最后，如果准备打开的文件没有匹配上 **auto-mode-alist** 里的任何一个正则表达式，那么 Emacs 就会把它放到 **default-major-mode** 变量的值所指定的那个主编辑模式里。这个模式通常被设置为基本模式，这是一个不带任何特殊功能的基本编辑模式。不过，有许多人喜欢把他们的默认编辑模式设置为文本模式，下面这条语句就是用来完成这件事的：

```
(setq default-major-mode 'text-mode)
```

虽然这一章介绍了不少对 Emacs 进行定制的好方法，但这些还只是一些皮毛。如果各位想做更进一步的研究，请进入第十三章学习 Emacs 中的 LISP 程序设计；如果想让 Emacs 完完全全地按照自己的意愿进行工作，打开这扇大门的钥匙就是 LISP 程序设计。



第十二章

程序员的 Emacs

本章内容：

- 语言编辑模式
- C 和 C++ 模式
- LISP 模式
- FORTRAN 模式
- 对程序进行编译

许多程序员都知道，程序设计工作通常可以分解为思考 - 编写 - 调试等三个环节组成的周期性循环。如果你曾经用 UNIX（或者其他操作系统）进行程序设计，那么你可能已经习惯于在程序开发周期的各个阶段，使用一组彼此互不衔接的工具；比如，一个用来写代码的文本编辑器、一个用来编译程序的编译器、再用操作系统本身来运行程序。如果有一种能够消除开发周期各阶段以及各阶段所用工具之间界线的开发环境，那么它无疑会提高效率。

Emacs 在程序的编写、运行和调试方面为很多种编程语言提供了强有力的支持，而且它把这种支持集成在一个平滑的框架里。既然在开发程序的时候不用退出 Emacs，我们就可以把注意力集中在真正的程序设计任务（即程序开发周期的“思考”部分）上，因为我们的空间将不再会浪费在挑选各种程序开发工具方面。

如果打算用 Emacs 来编写代码，那么可以利用 Emacs 针对某种程序设计语言的编辑模式来帮助更有效地完成任务。针对程序设计语言的编辑模式能够把 Emacs 从根本上改造成一个“语法指导的”或“语言敏感的”（即能够识别和理解某种程序设计语言的语法的）编辑器，从而帮助用户按自己的风格写出格式整齐、方便阅读的代码。Emacs 为很多种程序设计语言准备有相应的编辑模式。

Emacs 还支持程序的运行和调试工作。第五章介绍的 shell 模式允许把 Emacs 用做一种窗口系统，它使用户能够在编写代码的同时对它进行运行和调试。Emacs 还为很多种编译器和 UNIX 操作系统的 make 命令提供了一个功能强大的接口：Emacs 可

以对编译器给出的出错信息进行分析，并且能够根据出错语句的行号迅速到达文件中的出错地点。

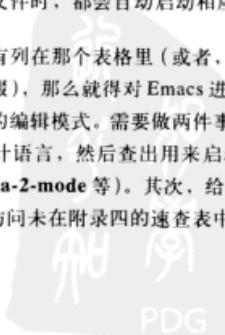
在这一章里，我们将对各种程序设计语言编辑模式的共同特点做概括性的探讨，然后有针对性地重点介绍三种程序设计语言——C 和 C++、LISP 以及 FORTRAN——的编辑模式。在讨论完 C/C++ 模式之后，我们还要介绍一下 **etags** 工具；对工作在文件数量众多的大型软件开发项目上的 C/C++ 程序员来说，这种工具能够带来极大的帮助。最后，我们将对 Emacs 与编译器 /make 命令之间的接口进行探讨。

语言编辑模式

我们已经见过一些 Emacs 的编辑模式了，比如文本模式（参见第二章和第八章）和 shell 模式（参见第五章）等。**list-buffer**（参见第四章）和 **dired**（参见第五章）等特殊功能其实也属于编辑模式的范畴。各种编辑模式都是由两大部分组成的：一个是用来实现这个编辑模式的 Emacs LISP 程序包，另一个是用来启动这个编辑模式的函数。

这本书主要以 Emacs Version 19 为讨论对象，这一版 Emacs 目前能够为以下一些程序设计语言提供相应的编辑模式：Ada、汇编语言、C、C++、Common LISP、FORTRAN、ICON、LISP、MIM、Modula-2、Objective-C、Pascal、Perl、PROLOG、Scheme、SGML 和 Simula，今后的版本还将会有所增加。有许多——但不是全部——语言编辑模式都在 Emacs 里“设好了挂钩”，如果你打开的文件带有适当的文件名后缀，Emacs 就会让你自动进入正确的编辑模式里。可以从附录四中的 Emacs LISP 程序包速查表里查出，Emacs 是否为所使用的程序设计语言准备了相应的编辑模式。如果表格最右一栏里列出了一个或更多的文件名后缀，就表示 Emacs 在遇到带有这些后缀名的文件时，都会自动启动相应的编辑模式。

如果文件名后缀没有列在那个表格里（或者，如果编译器允许使用没有在表格里列出的其他文件名后缀），那么就得对 Emacs 进行一些设置才能让它在访问源代码文件时自动启动相应的编辑模式。需要做两件事：首先，在程序包速查表的最右栏里找到使用的程序设计语言，然后查出用来启动对应编辑模式的函数的名字（比如 **ada-mode** 和 **modula-2-mode** 等）。其次，给 “*.emacs*” 文件增加几行代码，以便 Emacs 能够在用户访问未在附录四的速查表中，列出的文件时自动加载相应的程序包。



需要为此写两行代码。第一行代码使用 **autoload** 函数，其作用是告诉 Emacs 在遇到没有见过的命令时应该到什么地方去查它。这就在函数和实现函数的程序包之间建立起一个关联关系；这样，当这个函数第一次被调用时，Emacs 就会自动加载相应的程序包来载入该函数的代码。具体到我们的例子，需要在一个用来启动某语言编辑模式的函数和把实现该编辑模式的程序包之间建立起关联关系。**autoload** 函数的格式如下所示：

```
(autoload 'function "filename")
```

注意“function”前面的单引号和“filename”两端的双引号；LISP 语言的详细语法请参考第十三章内容。Ada 程序员可把下面这条语句添加到 “.emacs” 文件里：

```
(autoload 'ada-mode 'ada)
```

这条语句的作用是让 Emacs 在第一次调用 **ada-mode** 函数的时候加载 *ada* 程序包。

再看第二行代码，它将在源代码文件的后缀名和用来启动对应编辑模式的函数之间建立起一个关联关系；这样，当访问一个带有给定后缀名的文件时，那个用来启动对应编辑模式的函数将被自动调用执行。这需要用到第十一章里介绍的 Emacs 全局变量 **auto-mode-alist**。Emacs 将根据这个列表中的关联关系，把被访问的文件放到与它的后缀名相对应的编辑模式里。下面是一个为 **ada-mode** 创建这种关联关系的例子，Emacs 将把后缀名是 “.a” 的文件都放到这个编辑模式里（更进一步的解释请参考第十一章内容）：

```
(setq auto-mode-alist (cons '("." . ada-mode) auto-mode-alist))
```

这行 LISP 代码将在用户打开一个后缀名表明它是用某种编程语言写的源代码文件时，引发下述一连串事件。假设准备访问的文件是 “*pgm.a*”：首先，Emacs 读入这个文件。接着，它发现 **auto-mode-alist** 里有一个与文件名后缀 “.a” 对应的设置项，于是试图调用与之关联的 **ada-mode** 函数。它注意到函数 **ada-mode** 并不存在，但找到了一个把该函数与 *ada* 程序包关联起来的 **autoload** 设置项。于是，Emacs 将加载这个程序包，从中找到 **ada-mode** 命令，然后运行这个命令。经过这样一番周折之后，编辑缓冲区就被设置为 **ada-mode** 编辑模式。如果把刚才介绍的那两行代码放到了 “.emacs” 文件里，Emacs 就能在今后所有的编辑工作中识别出所使用的程序设计语言。



语法

不同的语言编辑模式在具体功能方面当然是有区别的，但它们都支持相同的基本概念。其中最重要的一点是，不同的语言编辑模式都能识别和理解与之对应的程序设计语言的语法，即能够识别和理解该语言的字符、词汇和某些语法现象。在对以前各章的讨论中，我们已经见识过 Emacs 在人类语言语法现象方面的“学问”了。在对普通文字材料做编辑处理的时候，Emacs 能够知道单词、句子和段落都是怎么一回事：它允许以这些语言学元素为单位，来移动光标或者删除文本。它还知道某些标点符号的正确用法，例如括号：只要一输入右括号，它就会“闪现”与之配对的左括号——把光标移到那里停一秒钟再返回来（注 1）。这为检查括号的配对情况是否正确提供了方便。

Emacs 对程序设计语言语法的了解，与它在人类语言语法方面的情况差不多。总的来说，它能够识别和理解以下一些基本的语言元素：

- 单词——对多数程序设计语言来说就是标识符和数字。
- 标点符号——包括操作符（如“+”、“-”、“<”、“>”等）和语句分隔符（如分号“;”）。
- 字符串——由字符组成的序列，通常放在一组配对的分隔符（比如引号）里。
- 括号——包括方括号（“[”和“]”）、花括号（“{”和“}”）和普通的圆括号。
- 空白——比如空格和制表符，用来分隔其他语言元素，通常不具备语言学方面的含义。
- 注释——被括在注释分隔符之间的字符序列，不同的语言有不同的注释分隔符（比如 C 语言中的“/*”和“*/”，LISP 语言中的分号“;”和换行符 LINEFEED）。

Emacs 把语法信息保存在各种“语法表（syntax tables）”里。类似于把按键绑定信息保存在键位映射图（keymap，参见第十一章）里的情况，Emacs 有一个供全体编

注 1：事实上，Emacs 向回寻找配对左括号的搜索距离是有一个最大限度（以字符为计算单位）的；它是变量 **blink-matching-paren-distance** 的值，默认设置为 12 000 个字符。左括号的“闪现”持续时间也是可设置的，它是变量 **blink-matching-delay** 的值（以秒为计算单位），默认设置为 1。

辑缓冲区使用的全局语法表，每个编辑缓冲区再有一个局部语法表，局部语法表会随编辑缓冲区所处的编辑模式而变化。此外，程序设计语言编辑模式还能识别和理解依赖于具体语言的更复杂的语言学概念，比如语句、语句块、函数、子例程和LISP中的S-表达式等等。

格式

除语言学方面的知识以外，Emacs的语言编辑模式还有许多能帮助写出格式整齐的代码的功能。这些功能涉及到缩进、注释等与程序设计风格有关的各个方面，从而保证程序有良好的一致性和可读性，让注释排列的整齐划一等等。更重要的是它们能够减轻工作负担，用户不用再去操心语句的缩进量是否正确，甚至不用记住当前的缩进量是多少。这些功能最好的地方是它们基本上都可以由用户来自行定制。

我们在文本模式里见过这样的情况：如果在行尾输入的是换行符**LINEFEED**（或者“**C-j**”组合键）而不是回车键，Emacs就会自动对下一行进行正确的缩进。这种缩进由变量**left-margin**控制，这个变量的取值就是以字符计算的文本缩进量。程序设计语言的编辑模式也具备相同的功能，只是这种处理将更加灵活和复杂。

类似于文本模式中的情况，语言编辑模式里的**LINEFEED**也会对下一行进行正确的缩进。也可以在语句输入完毕之后再对它进行缩进，即把光标放在这一行的任意位置再按下**TAB**键即可。

有些语言编辑模式还为代码语句的结束符（比如分号或者右花括号）特别准备了一项功能：输入这些字符的时候，Emacs将自动对当前行进行缩进。Emacs文档把这种行为叫做“自动缩进”（英文原文是“electric”）。大多数语言编辑模式还提供有成套的变量组合来控制缩进样式（这些变量也都能够进行定制）。

表12-1列出了其他一些与缩进有关的命令，它们的具体工作情况是由具体程序设计语言的有关规则来决定的。

表12-1：基本缩进命令速查表

键盘操作	命令名称	动作
ECS C-\	<code>indent-region</code>	对光标和文本块标记之间的每一行进行缩进

表 12-1：基本缩进命令速查表（续）

键盘操作	命令名称	动作
ECS m	back-to-indentation	把光标移到当前行的第一个非空白字符上
ECS ^	delete-indentation	把当前行合并到上一行去

下面是一个“ECS C-\”命令的用法示例。例子中的程序代码是用C语言写的，我们以后的例子也会用到这段程序。本节各示例所阐明的概念也适用于大多数其他程序设计语言中；LISP和FORTRAN语言中与此相当的操作将在讨论其语言编辑模式的章节里进行介绍。

假设已经输入下面这段C语言代码：

```
int times (x, y)
int x, y;
{
int i;
int result = 0;

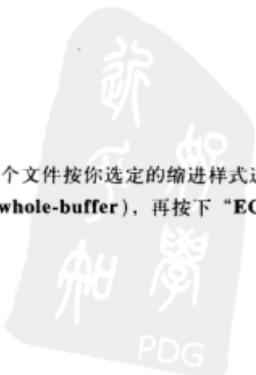
for (i = 0; i < x; i++)
{
result += y;
}
}
```

现在，把文本块标记设置在这段代码的开始，把光标放在它的末尾，然后按下“ECS C-\”组合键。Emacs将把它排成如下所示的样子：

```
int times (x, y)
  int x, y;
{
  int i;
  int result = 0;

  for (i = 0; i < x; i++)
  {
    result += y;
  }
}
```

也可以方便地用“ECS C-\”组合键把整个文件按你选定的缩进样式进行缩进：先按下“C-x h”组合键（命令名是**mark-whole-buffer**），再按下“ECS C-\”组合键即可。



“**ECS m**”命令可以迅速把光标移动到某行代码真正开始的地方。请看下面的例子。假设光标现在的位置是：

```
int result = 0;
```

如果你按下“**ECS m**”组合键，则光标将移动到“int”的第一个字母上，如下所示：

```
int result = 0;
```

我们来看一个使用“**ESC ^**”命令的例子。假设想让 for 语句的左花括号与关键字 for 出现在同一行上。把光标放在这个左花括号所在文本行上的任意位置，按下“**ESC ^**”组合键，则该语句将变成如下所示的样子：

```
for (i = 0; i < x; i++) {  
    result += y;  
}
```

语言编辑模式通常还特别提供一些与该语言某些特定功能有关的缩进命令。

因为程序设计语言全都有注释语法，所以 Emacs 准备了几个通用的注释处理命令：它们在不同的语言编辑模式里，会根据该语言的具体规定产生不同的效果。适用于各种语言编辑模式的通用注释命令是“**ESC ;**”（命令名是 `indent-for-comment`）（注 2）。按下“**ESC ;**”组合键的时候，Emacs 将按变量 `comment-column` 的值进行缩进；如果这一行上的文本已经超出了这一列的位置，它将从最后一个字符的位置再往右移一个空格。然后，Emacs 插入一个注释分隔符（或者一组配对的注释分隔符，比如 C 语言中的“`/*`”和“`*/`”），再把光标放在左分隔符的后面。

请看下面的例子。如果想给上面那个函数中 for 循环的循环体语句加上一条注释，请把光标放的这个语句所在行的任意位置上，再按下“**ESC ;**”组合键，结果将是：

```
result += y; /* ■ */
```

这就能够顺势把注释输入到两个分隔符之间。如果想对一个更长的语句行做同样的事，比如下面这条语句：

```
a_i = term_arr[i].num_docs / total_docs;
```

注 2：LISP 程序员应该比较容易记住这个命令，因为 LISP 语言中的注释就是以分号（`;`）开始的。

则结果将是：

```
result += y; /* add the multiplicand */
```

comment-column 变量是可以定制的，只要把相应的代码添加到 “.emacs” 文件里即可（如果想对该变量做永久性设置，这是最有效的方法）。但如果只是想在当前编辑缓冲区里对 **comment-column** 变量做临时性的设置，那么只需把光标移动到想让注释开始的位置再按下 “C-x ;” 组合键（命令名是 **set-comment-column**）即可。注意，这个命令只影响当前编辑缓冲区里的 **comment-column** 变量的值；而该变量在其他编辑缓冲区（即使是同一编辑模式下的其他编辑缓冲区）里的值却不会改变。

如果注释的内容用一个文本行容纳不下，就需要在下一行继续写完注释：“**ESC j**”组合键（命令名是 **indent-new-comment-line**）就是用来做这件事的。这个命令将在下一行开始一条新的注释（有些语言编辑模式允许把它定制为继续刚才的注释继续往下写，而不是另外开始一个新的注释行）。假设已经给 “times” 函数里 for 循环体中的语句输入了一些注释（但还没有输入完）；光标目前停在文本的末尾处，如下所示：

```
result += y; /* add the multiplicand */
```

现在，想在第二行上继续输入注释。如果按下 “**ESC j**” 组合键，就将得到如下所示的效果：

```
result += y; /* add the multiplicand */  
/* | */
```

接下来，可以立刻开始输入第二行注释。也可以用 “**ESC j**” 命令把现有的注释文本拆分为两行。假设光标是在如下所示的位置：

```
result += y; /* add the multiplicand */
```

如果此时按下 “**ESC j**” 组合键，结果将是：

```
result += y; /* add the */  
/* | multiplicand */
```

如果想把一小段代码整个地设置为注释，可以使用 **comment-region** 命令（除在少数几种语言编辑模式里以外，这个命令没有绑定到任何组合键上）。假设有一段下面这样的代码：



```
this = is (a);
section (of, source, code);
that += (takes[up]->a * number);
of (lines);
```

像平常那样把这段代码定义为一个文本块，然后输入“**ESC x comment-region**”命令，结果将是：

```
/*      this = is (a); */
/*      section (of, source, code); */
/*      that += (takes[up]->a * number); */
/*      of (lines); */
```

“**ESC x kill-comment RETURN**”命令（它也没有默认的按键绑定）可以把当前行上的注释都清除掉，很适合用来清除程序中自成一行的“单行”注释。光标不必非得处于注释文本之中。每一种语言编辑模式都有一些与该语言注释密切相关的功能，其中往往包括有用来对缩进样式进行定制的各种变量。

在介绍完适用于各种语言编辑模式的通用概念之后，我们将对 C、C++、LISP 和 RORTAN 等程序设计语言的编辑模式做专门的探讨。随后的三个小节将对这几种语言编辑模式依次进行讨论；在关于 C 和 C++ 语言编辑模式的小节末尾，我们还要研究一下 **etags** 工具，它对编写大型、多文件程序的 C / C++ 程序员有很大的帮助。

如果你只对一两种程序设计语言感兴趣，就没有必要阅读以下所有的内容。如果你是用其他程序设计语言来写程序的，而 Emacs 又恰好提供有相应的语言编辑模式，那你可以从下面这三个小节里挑选一个阅读，体会一下语言编辑模式的“味道”；虽然具体的语言编辑模式会有所不同，但都具有相同的基本概念，起码能让你开个好头。

C 和 C++ 模式

如果打开后缀名是 “.c”、“.h”、“.y”（用在 **yacc** 语法方面）或 “.lex”（lex 定义文件）的文件，Emacs 就会自动进入 C 语言编辑模式，简称 C 模式。如果打开后缀名是 “.C”、“.H”、“.cc”、“.hh”、“.cpp”、“.cxx”、“.hxx”、“.c++” 或 “.h++” 的文件，Emacs 就会自动进入 C++ 语言编辑模式，简称 C++ 模式。还可以利用“**ESC x c-mode RETURN**”命令把任意文件手动地放到 C 模式里。类似地，“**c++mode**”命令将把编辑缓冲区放到 C++ 模式里。

C 模式和 C++ 模式是用同一个 Emacs LISP 程序包实现的，它的名字是“cc-mode”；除了这两种语言编辑模式，该程序包还提供一个与用在 NextStep 环境里的 Objective-C 语言相对应的编辑模式。C 模式能够识别和理解 ANSI C 语法和“古老的” Kernighan-Ritchie C 语法。虽然我们的介绍以 C 模式为主，但它们全都适用于 C++ 模式。C++ 模式只比 C 模式额外多出很少几项功能。我们将在这一小节的末尾对它们进行探讨。

顺便说一句，Emacs 的 Perl 语言编辑模式（这种程序设计语言由于广泛应用于 WWW 服务器的程序设计中而正变得越来越流行）是从 C 模式的一个老版本演化而来的。如果你使用的程序设计语言是 Perl，就会发现 C 模式里的光标移动命令、缩进命令和排版命令等都能原封不动地应用到 Perl 模式里，惟一需要变化的就是得把命令名中的“c-”前缀改为“perl-”。Emacs 会在遇到后缀名为“.pl”的文件时自动启动 Perl 模式。

在 C 模式里，Emacs 能够识别和理解本章前面提到的各种语言学元素。语句中的分号（;）、冒号（:）、逗号（,）、花括号（{ 和 }）以及井字号（#，用在 C 语言的预处理器命令中）等字符都是自动缩进的。也就是说，输入这些字符的时候，Emacs 将自动对当前行进行缩进。

移动命令

除了以单词、句子等语言元素为单位进行移动的 Emacs 标准命令（这些命令在 C 模式里的用武之地，可以说只剩下大段的注释了）之外，C 模式还另外准备了一些能够以语句、函数（注 3）以及预处理器条件为单位进行移动的高级命令。我们把 C 模式这些高级的移动命令汇总在表 12-2 里。

表 12-2：C 模式高级移动命令速查表

键盘操作	命令名称	动作
ESC a	c-beginning-of-statement	移动到当前语句的开头
ESC e	c-end-of-statement	移动到当前语句的末尾

注 3：与 C 语言函数有关的 Emacs 编辑命令在它们的命令名里都有“defun”字样，这是从 LISP 模式借用过来的用法。在 LISP 语言里，一个 defun 就是一个函数定义。

表 12-2: C 模式高级移动命令速查表 (续)

键盘操作	命令名称	动作
ESC q	c-fill-paragraph	如果光标在注释文本中间，则进行段落重排，保留缩进和前导字符
ESC C-a	beginning-of-defun	移动到光标所在函数的开头
ESC C-e	end-of-defun	移动到光标所在函数的末尾
ESC C-h	c-mark-function	把光标放到函数的开头，把文本块标记放到函数的末尾——即把函数整个选取为一个文本块
C-c C-q	c-indent-defun	按缩进样式对整个函数进行缩进
C-c C-u	c-up-conditional	移动到当前预处理器条件的开始位置
C-c C-p	c-backward-conditional	移动到上一个预处理器条件
C-c C-n	c-forward-conditional	移动到下一个预处理器条件

请注意，以语句为单位的移动命令其按键绑定与 Emacs 的 **backward-sentence** 和 **forward-sentence** 命令的完全相同。事实上，在 C 语言的注释里，它们的使用效果完全相同。

类似地，“**ESC q**”正是原来的 **fill-paragraph** 命令；只不过 C 模式把它的功能扩大为能够保留语句开头的缩进和前导字符而已。举个例子，如果光标位于下面这段注释里的某个任意位置：

```
/* This is
 * a
 * comment paragraph with wildly differing right
 * margins.
 * It goes on      for a while,
 * then stops.
 */
```

那么，按下“**ESC q**”组合键时，它将变成下面这样：

```
/* This is a comment paragraph with wildly differing right margins.
 * It goes on for a while, then stops. */
```

如果不得不通读别人的大段代码以对之进行调试或者修改，就会发现以预处理器条件为单位的移动命令是一种上帝的恩赐。当面对的代码是准备运行在多种系统

(Emacs就是一个这样的软件)上的时候,情况就更是如此;对于这种情况,你最需要答案的问题是:“到底有哪些代码是真正会被编译的?”

有了“**C-c C-u**”命令,就能立即找出某行代码具体是由哪一个预处理器条件控制的。请看下面的例子:

```
#define LUCYX
#define BADEXIT -1

#ifndef LUCYX
...
*ptyv = open ("/dev/ptc", O_RDWR | O_NDELAY, 0);
if (fd < 0)
    return BADEXIT;
...
#else
...
fprintf (stderr, "You can't do that on this system!");
...
#endif
```

假设代码里的省略号 (...) 都代表上百行代码,而现在需要找出文件 /dev/ptc 在什么条件下才会被打开。于是,把光标移到这一行上并按下“**C-c C-u**”组合键,光标将移动到“#ifndef LUCYX”语句处:表示它只有在 LUCYX 系统上时才会被编译。如果想跳过不会被编译的代码,而直接到达这个编译条件的结尾,请按下“**C-c C-n**”组合键。我们在本小节的后面还会再介绍一个与 C 语言预处理器有关的命令。

C 语言中的语句分隔符和语句块分隔符被绑定为一组不仅能正确插入字符,还能正确缩进有关语句的命令上。这些字符是左花括号 “{”、右花括号 “}”、分号 “;” 和冒号 “:”(用来分隔 switch 语句的标号)。比如,如果准备结束一个语句块或者一个函数体,就可以按下 **LINEFEED** (或 **RETURN**) 和 “)”; Emacs 将把它和与它配对的 “{” 对齐,也就不用再去查看前面的代码以确定那个 “{” 到底是在哪一列上了。

“)”是一种括号字符,所以 Emacs 会在输入 “)”的时候“闪现”与它配对的 “{”。如果配对的 “{” 不在窗口所显示的文本范围内,Emacs 就将把包含 “{” 的那条语句显示在辅助输入区里;如果那一行除 “{” 外只有空白(即空格或制表符),Emacs 会先显示一个 “^J”(即 “**C-j**” 或 **LINEFEED** 字符),然后把下一行显示出来,这是为了让用户更好地把握 “{” 字符处的上下文。

我们再去看看前面给出的“times”函数的例子。假设准备在这个函数的结尾输入一个“}”字符，而这个函数体开始位置上的“(”字符已经不在屏幕上。因为在左花括号“(”所在的那一行上没有其他的代码，所以输入“)”后会在辅助输入区里看到如下所示的内容：

```
Matches { ^J int i;
```

对代码的缩进样式进行定制

C 语言（或者其他程序设计语言）的编程风格（coding style）是很个性化的。C 程序员是通过各种书籍或者别人写的各种代码片段来学习这种语言的，但他们迟早会形成自己的风格，而这些个性化的风格不见得会与他们在学习时接触到的一模一样。

C 模式的缩进行为恰恰是这种语言学习手段的体现，它还提供很多用来对其缩进行为进行定制的功能。在最简单的层次上，可以挑选一种现成的缩进样式来使用；然后，如果不满意，就可以对挑选出来的样式进行定制，甚至可以从点滴开始创建自己的样式。后一种做法需要对 Emacs LISP 程序设计知识（参见第十三章）有比较深的掌握，还需要有一点勇敢精神。

“**ESC x c-set-style**”命令的作用就是让从现成的缩进样式里挑出一个来使用。这个命令会提示用户输入自己想要的缩进样式的名称，此时最简单的方法是按下**TAB** 键——Emacs 的自动补足命令（参见第六章），它将打开一个“*Completions*”编辑缓冲区，那里面将列出所有的可选项。如果想选择某种缩进样式，输入它的名称再按下次回车键即可。

Emacs 已经为我们提前预备了一些缩进样式，它在默认情况下加载的缩进样式见表 12-3 所示。

表 12-3：C 模式缩进样式速查表

缩进样式	说明
CC-MODE	默认的编程风格；其他样式都是从它推导出来的
GNU	Emacs 自身和其他 GNU 程序所使用的 C 语言书写风格
K&R	Kernighan 和 Ritchie 合著的《The C Programming Language》一书中使用的编程风格，这本书是 C 语言的开山之作

表 12-3: C 模式缩进样式速查表 (续)

缩进样式	说明
BSD	BSD 系列的 UNIX 版本使用的编程风格
Stroustrup	Bjarne Stroustrup 所著的《The C++ Programming Language》一书中使用的编程风格，这本书是 C++ 语言的标准参考书
Whitesmith	Whitesmith 公司在他们的 C 和 C++ 编译器软件的文档里使用的编程风格
Ellemtel	瑞典 Ellemtel 电讯系统实验室的 C++ 文档里使用的编程风格

为了让大家对这些编程风格有一些了解，我们用本章前面给出的 C 函数为例做一下演示：

```
int times (x, y)
int x, y;
{
    int i;
    int result = 0;

    for (i = 0; i < x; i++)
    {
        result += y;
    }
}
```

如果把这段代码选取为一个文本块，并按下“**ESC C-v**”组合键（命令名是 **indent-region**），那么 Emacs 将按其默认的缩进样式，把这段代码重新排版为下面这个样子：

```
int times (x, y)
int x, y;
{
    int i;
    int result = 0;

    for (i = 0; i < x; i++)
    {
        result += y;
    }
}
```

用“**ESC x c-set-style**”命令选择使用“**K&R**”风格，然后对段落进行重排，代码将排列为下面这个样子：



```
int times (x, y)
int x, y;
{
    int i;
    int result = 0;

    for (i = 0; i < x; i++)
    {
        result += y;
    }
}
```

如果想切换到GNU风格的缩进样式，请在选中风格“GNU”后对代码块进行重排；结果将是：

```
int times (x, y)
    int x, y;
{
    int i;
    int result = 0;

    for (i = 0; i < x; i++)
    {
        result += y;
    }
}
```

选好编程风格之后，把下面这条语句添加到“*.emacs*”文件里就可以永久地设置它：

```
(add-hook 'c-mode-hook
          '(lambda ()
            (c-set-style 'stylename ')))
```

这条语句到底做了些什么还得等学到下一章时才能讲明白。就目前而言，大家只要别忘了在第2行的“*(lambda*”的前面加上一个单引号就可以了。

每种编程风格都有它自己的特点，而让Emacs把这些特点都实现出来，可并不是件轻而易举的事情。虽然Emacs的早期版本定义了几个用来控制缩进层次的变量，但数量并不是很多；它们不仅不容易使用，而且说老实话，也很难百分之百地从细节上区分出不同的编程风格。与此形成鲜明对比的是，Emacs最新版本里的C模式使用了一个规模相当巨大的变量集。这个变量集实在是太大了，除了那些爱钻牛角尖的Emacs LISP高手之外，一般人根本对付不了。

换句话说，如今的 C 模式是靠把成组的变量，和它们的值分门别类地归纳为缩进样式而实现出各种编程风格的。Emacs 用了一个非常庞大的变量（它的名字是 `c-style-alist`）来容纳全体缩进样式和它们的关联信息。各位可以通过修改现有缩进样式里的变量值，或者自行添加缩进样式这两种方法来对这个大家伙进行定制。具体细节请查阅自己系统的 Emacs LISP 目录里的 `cc-mode.el` 文件（详见第十三章）。

C 和 C++ 模式的附加功能

C 模式还有很多非常有价值的功能，有的用途广泛，有的仅适用于极其罕见的情况。自动开始新行（auto-newline）和“饥饿的删除键”（hungry-delete-key）可以说是这些功能中最有意思的两个了，它们是两种用来给按键操作增加自动缩进功能的办法（注 4）。

启用了自动开始新行功能之后，只要一输入分号 “;”、花括号 “(” 和 “)” 或者（在某些特定场合下）逗号 “,” 及冒号 “：“，Emacs 就会自动加上一个换行符并对新行做正确的缩进。这项功能可以节省一些时间，而且能够帮助保持代码书写风格的一致性。

自动开始新行功能在默认的情况下是处于禁用状态的；可以用“**C-c C-a**”组合键（命令名是 `c-toggle-auto-state`）启用它。（重复这个命令将再次禁用这项功能）。状态行上的“(C)”标志改变为“(C/a)”。我们以 `times()` 函数为例来看看它究竟是怎样工作的。先输入头两行代码，当输入了第 2 行上的“y”时，我们看到的是：

```
int times (x, y)
int x, y
```

现在，按下分号键；Emacs 将插入一个换行符并把光标移到下一行，如下所示：

```
int times (x, y)
int x, y
```

输入左花括号，同样的事情将再次发生，如下所示：

注 4： 它们可以用来仿真早期 Gosling Emacs 中的 `electric-c-mode` 模式。

```
int times (x, y)
int x, y;
{
```

■

当然，Emacs 在输入 “(” 之后的缩进量要取决于正在使用的缩进样式。

另一种可选用的自动缩进功能即“饥饿的删除键”，它在默认的情况下也是处于禁用状态的。可以用“**C-c C-d**”组合键（命令名是 **c-toggle-hungry-state**）启用之。状态行上的“(C)”标志改变为“(C/h)；或者，如果已经启用了自动开始新行功能，状态行上的“(C/a)”标志将改变为“(C/ah)”。

“饥饿的删除键”功能将使 **DEL** 键具备删除光标左面全部空白的能力。我们还用刚才的例子接着往下说。假设刚输入那个左花括号。现在，如果按下 **DEL** 键，则 Emacs 将一直删到这个左花括号处，如下所示：

```
int times (x, y)
int x, y;
```

■

自动开始新行和“饥饿的删除键”这两种功能的启用 / 禁用状态可以同时用“**C-c C-t**”组合键（命令名是 **c-toggle-auto-hungry-state**）进行切换。

如果让这两项功能中的某一种在启动 Emacs 的时候自动处于启用状态，就需要把下面这样的语句添加到 **.emacs** 文件里：

```
(add-hook 'c-mode-hook
          '(lambda ()
            (c-toggle-auto-state)))
```

如果想用另外的 C 模式定制来合并这个定制（如前例中的缩进样式），就需要将 Emacs LISP 代码行合并如下：

```
(add-hook 'c-mode-hook
          '(lambda ()
            (c-set-style "stylename")
            (c-toggle-auto-state)))
```

这条关联语句的含义将在第十三章里的“对现有编辑模式进行定制”一节里进行探讨。



C 模式还提供一些与程序注释有关的支持功能。在本章前面的内容里，我们已经对这方面的支持功能做过概括性的讨论，但并没有对某种编辑模式在这方面的具体功能做细致的说明。大家可以对“**ESC j**”（命令名是 **indent-new-comment-line**）的执行情况进行定制，从而使 Emacs 在下一行继续同一条注释，而不是另外创建一组新的注释分隔符。这项功能是由变量 **comment-multi-line** 控制的：如果它设置为“**nil**”（这是它的默认值），Emacs 就将在下一行创建一条新的注释，就像在本章前面的示例中所给出的那样，如下所示：

```
result += y;          /* add the multiplicand */  
/* █ /
```

在输入单词“multiplicand”之后，立刻按下“**ESC j**”组合键就会得到这样的结果；现在的光标位置表示可以立刻开始输入第二个注释行上的文本。如果把变量 **comment-multi-line** 设置为“**t**”（或者任意一个不是“**nil**”的值），将得到如下所示的结果：

```
result += y;          /* add the multiplicand  
█ /
```

我们将要介绍的最后一个功能是“**C-c C-e**”组合键（命令名是 **c-macro-expand**）。和以预处理器条件为单位的移动命令（比如绑定在“**C-c C-u**”组合键上的 **c-up-conditional** 命令）一样，**c-macro-expand** 命令也能帮助回答“到底有哪些代码是真正会被编译的？”这个通常很难回答的问题。

在使用 **c-macro-expand** 命令之前，必须先定义一个文本块。接着，按下“**C-c C-e**”组合键时，这个文本块中的代码将被送往实际的 C 语言预处理器去进行处理，其结果将被放到一个名为“*Macroexpansion*”的窗口里。

我们用本章前面那个带有 C 预处理器指令的代码为例，来看看这一过程的具体工作情况，如下所示：

```
#define LUCYX  
#define BADEXIT -1  
  
#ifdef LUCYX  
    *ptyv = open ('/dev/ptc', O_RDWR | O_NDELAY, 0);  
    if (fd < 0)  
        return BADEXIT;  
#else
```

```
    fprintf (stderr, "You can't do that on this system!");  
#endif
```

如果把这段代码定义为一个文本块并按下“**C-c C-e**”组合键，那么就会出现下面这条消息：

```
Invoking /lib/cpp -C on region . . .
```

过一会儿，出现

```
done
```

接下来，出现一个“*Macroexpansion*”窗口，窗口里是这次处理的结果：

```
*ptyv = open ("/dev/ptc", O_RDWR | O_NDELAY, 0);  
if (fd < 0)  
    return -1;
```

如果想让 **c-macro-expand** 调用另外一个 C 预处理器命令而不是默认的 “/lib/cpp -C”（选项 “-C” 表示“把注释保留在输出里”），就需要对变量 **c-macro-preprocessor** 做相应的设置。比如，如果想试用文件名是 “/usr/local/lib/cpp”的预处理器，就需要把下面这条语句添加到 “.emacs” 文件里：

```
(setq c-macro-preprocessor "/usr/local/lib/cpp -C")
```

我们强烈建议大家保留 “-C” 选项以免预处理器删掉代码中的注释。

C++ 模式的差异

我们在前面已经说过，C++ 模式与 C 模式使用的是同一个 Emacs LISP 程序包。在 C++ 模式里的时候，Emacs 将使用 C++ 语法，而不再是 C 语言（或 Objective-C 语言）的语法。我们刚才介绍的命令中有几个会因此而导致其行为发生变化。

C++ 模式和 C 模式并没有明显的差异。它们之间最重要的区别在于为了定制 C++ 模式，而必须添加到 “.emacs” 文件里的 Emacs LISP 代码：必须把语句中的 “**c-mode-hook**” 字样替代为 “**c++-mode-hook**”。比如说，如果想把 C++ 模式的缩进样式从默认情况设置为 Stroustrup，就必须把下面这条语句添加到 “.emacs” 文件里：

```
(add-hook 'c++-mode-hook  
         '(lambda ()
```

```
(c-set-style 'Stroustrup )
(c-toggle-auto-state))
```

请注意，可以用这种方法为C模式和C++模式分别设置其各自的关联关系；也就是说，如果你会同时使用这两种程序设计语言，就完全可以给它们分别设置不同的缩进样式。

C++模式还提供了一个附的命令“**C-c :**”（命令名是**c-scope-operator**）。这个命令的作用是插入C++语言中的域操作符——双冒号（::）。之所以需要有这样一个命令是因为单冒号（:）通常会绑定有自动缩进功能——输入冒号就会对代码进行缩进，可本意却并非如此。域操作符可能出现在C++代码中的任何位置，而单冒号却往往用来指示一个case语句的标号，因此它需要特殊的缩进。“**C-c :**”命令乍看起来好像有些多余，但它却是为了避免C++语言中的语法冲突所必须的折中办法。

最后，C模式和C++模式中都有**c-forward-into-nomenclature**和**c-backward-into-nomenclature**命令，它们在默认的情况下没有绑定到任何按键上。这两个命令与**forward-word**和**backward-word**命令分别有几分相似，但它们会把单词中间的大写字母看做是新单词的开始。比如说，它们会把“*ThisVariableName*”看做是3个单词，而标准的**forward-word**和**backward-word**命令会把它看做一个单词。用“*ThisTypeOfVariableName*”形式的变量名是C++程序员比较习惯的编程风格，与此相对的是“*this_type_of_variable_name*”，这种形式的变量名是C语言代码里比较常见的。

C++程序员可能需要把**c-forward-into-nomenclature**和**c-backward-into-nomenclature**命令绑定到原来绑定为标准的单词移动命令的按键组合上。我们将在第十三章的“对现有编辑模式进行定制”一节介绍有关的具体做法。

我们已经把C模式和C++模式的主要功能介绍得差不多了，但这两种编辑模式还有很多我们没有讲到的其他功能。这些功能中有很多是一般人很少会用到的，只有那些专家级的Emacs LISP定制者才有可能需要它们。详细情况请参考Emacs LISP程序包**cc-mode.el**，它可是有一个将近5 000行代码的家伙。



etags

Emacs 为 C 和 C++ 程序员准备的另外一个功能是 **etags** 工具程序（注 5）。**etags** 也能用来对付许多以其他程序设计语言写成的代码，其中包括 FORTRAN、Perl、Pascal、LATEX、LISP 以及多种汇编语言。如果你的软件项目规模较大，涉及到的文件也比较多，就会发现 **etags** 确实会给你带来非常大的帮助。

从原理上讲，**etags** 不过是一个面向多文件的搜索工具，它既了解 C 语言和 Perl 语言中的函数定义结构，又具备通用性的查找功能。有了它，就能从整个目录的范围内把某个函数找出来而无需记住定义此函数的文件；查找和查询—替换操作也将延伸到多个文件。**etags** 使用“函数标签表（tag tables）”，其中包含着对应于目录中每个文件的函数名单，和各个函数的定义在文件的定义位置等信息。与 **etags** 有关的许多命令都需要在查找字符串里使用正则表达式（参见第三章）。

为了使用 **etags** 工具，需要先在当前目录里调用运行 **etags** 程序以创建函数标签表，这个程序的参数是打算收集其函数标签信息的文件。最常见的情况是用“**etags *.[ch]**”命令来调用 **etags** 程序，意思是为全体以“.c”和“.h”结尾的文件建立一个函数标签表。**etags** 程序的输出（也就是函数标签表）将保存在一个名为 **TAGS** 的文件里。在书写代码的时候，可以随时调用 **etags** 程序刷新函数标签表，让它能及时反映出新文件里的新函数定义。

建立了函数标签表之后，还需要让 Emacs 知道它：“**ESC x visit-tags-table RETURN**”命令就是用来完成这项工作的。它会提示输入函数标签表文件的名字——默认为当前目录里的 **TAGS** 文件。完成了这个步骤之后，就可以使用各种各样的 Emacs 函数标签命令了。

最重要的函数标签命令是“**ESC .**”（命令名是 **find-tag**）。这个命令提示输入一个用来在函数标签表里查找某个函数的字符串，而这个字符串会出现在某个函数的名字里。给出查找字符串，Emacs 将找出与之匹配的函数名，并把包含该函数的文件打开在当前窗口里，并且会移动到该函数定义部分的第一行。“**ESC .**”有一个变体是“**C-x 4 .**”（命令名是 **find-tag-other-window**），它将另外使用一个窗口而不是替换当前窗口里的文本。

注 5： ex 和 vi 用户可以把 **etags** 看做是 **ctags** 的一个功能更为强大的“近亲”。

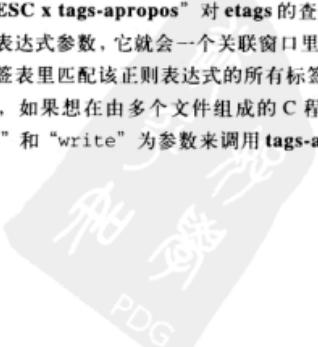
“**ESC .**”的特点是能把光标位置上的单词用做默认的查找字符串。比如说，如果光标在字符串“`my_function`”的某个字母上，“**ESC .**”就会把“`my_function`”当作默认的查找字符串。换句话说，如果你正在读一条调用某个函数的C语句，直接按下“**ESC .**”组合键就能查看到该函数的代码。

如果好几个函数同名，“**ESC .**”找出的将是按字母表顺序最先出现的那个函数。要想查找其他的同名函数，可以用“**ESC ,**”（命令名是**tags-loop-continue**）找到下一个（如果找不到下一个，Emacs将报告出错）。这项功能特别适用于子目录里有不止一个程序的情况；也就是说，如果有不止一个函数叫做“`main`”，“**ESC ,**”组合键就会有其他的用法。我们马上就会讲到。

函数标签表还可以用来查找函数定义以外的其他东西。命令“**ESC x tags-search RETURN**”将提示输入一个正则表达式；它将以列在函数标签表里的全体文件（比如所有的“.c”文件和“.h”文件）为对象查找有无与该正则表达式匹配的事物，不论它是否是一个函数。这项功能与我们将在本章结尾处介绍的**grep**工具程序有异曲同工之效。在调用执行“**tags-search**”命令之后，能够用按下“**ESC ,**”组合键的方法把其他的匹配情况都找出来。

etags还有一项能够仿真查询 - 替换操作的功能。命令“**ESC x tags-query-replace RETURN**”将以列在函数标签表里的全体文件为对象做一次正则表达式查询 - 替换操作。对正则表达式查询 - 替换操作的讨论请参考第三章。类似于正常的“**query-replace-regexp**”命令，如果给**tags-query-replace**加上一个前缀（比如像“**C-u ESC x tags-query-replace RETURN**”那样），Emacs就只替换与整个单词完全匹配的情况。如果想在不影响“`fprintf`”语句的前提下替换“`printf`”，这个功能就非常有用。如果用“**ESC**”键或“**C-g**”组合键退出了一次**tags-query-replace**操作，就还可以通过按下“**ESC ,**”的方法从刚才的退出位置重新继续往下进行。

命令“**ESC x tags-apropos**”对**etags**的查找功能做了进一步的补充。如果给它一个正则表达式参数，它就会在一个关联窗口里打开一个“*Tags List*”编辑缓冲区，函数标签表里匹配该正则表达式的所有标签（包括文件名和函数名）都列在其中。比如说，如果想在由多个文件组成的C程序里找出输出例程的名字，就可以用“`print`”和“`write`”为参数来调用**tags-apropos**命令。



最后，可以用命令“**ESC x list-tags RETURN**”把函数标签表里的全体标签——即某个给定 C 文件的全体函数都列出来。在提示符处给出文件名，就会在“*Tags List*”编辑缓冲区里看到在该文件里定义的全部函数的名称及其返回值的类型(如果有)。如果把光标移动到这个名单里，就可以用“**ESC .**”组合键来查看函数的实际代码。这是因为“**ESC .**”能够把光标位置上的单词用做默认的函数名称，只需把光标移动到想查看的函数的名称上再按下“**ESC .**”和回车键就可以看到它的具体内容。

LISP 模式

总共有 3 种 LISP 编辑模式，我们按它们“**ESC x**”命令名称的先后顺序列在下面：

emacs-lisp-mode

用来编辑 Emacs LISP 代码，请参考第十三章中的讨论(文件名以“*.emacs*”或“*.el*”结尾)。

lisp-mode

用来编辑供其他 LISP 系统使用的 LISP 代码(文件名以“*.l*”或“*.lisp*”结尾)。

lisp-interaction-mode

用来编辑和运行 Emacs LISP 代码。

这 3 种编辑模式的基本功能都是一样的，它们之间的区别体现在它们各自对 LISP 代码运行的支持功能上。

这 3 种编辑模式能够识别和理解各种语言编辑模式所共有的基本语法元素。除此之外，它们还专门为 S- 表达式、列表和函数定义等高级语法概念准备了各种命令。S- 表达式(syntactic expression, 语法表达式)指的是任意一个没有语法错误的 LISP 表达式，它可以是一个原子项(数字、符号、变量，等等)或者一个放在括号里的列表。列表(list)是 S- 表达式的特殊形式，而函数定义(defun)又是列表的特殊形式。这些 LISP 语言所独有的语法概念需要一些专门的命令来处理。对一般的使用情况而言，只要掌握这些命令的一个子集就足能应付了。

表 12-4 列出了对 S- 表达式进行处理的有关命令。



表 12-4: S- 表达式处理命令速查表

键盘操作	命令名称	动作
ESC C-b	backward-sexp	移动到上一个 S- 表达式
ESC C-f	forward-sexp	移动到下一个 S- 表达式
ESC C-t	transpose-sexps	交换光标前后的两个 S- 表达式的位置
ESC C-@	mark-sexp	把文本块标记设置在当前 S- 表达式的末尾，把光标设置在当前 S- 表达式的开头
ESC C-k	kill-sexp	删除光标后面的那个 S- 表达式
(无)	backward-kill-sexp	删除光标前面的那个 S- 表达式

因为 S- 表达式可以是各种各样的东西，所以对 S- 表达式进行处理的命令到底会采取什么动作，要取决于调用命令时光标的的具体位置。如果光标在一个左括号 “(” 或它前面的一个空白位置上，将被处理的 S- 表达式就是以这个左括号 “(” 开始的列表。如果光标是在字母或数字等其他字符（或该字母或数字前面的一个空白位置）上，将被处理的 S- 表达式就是一个原子项（符号、变量或常数等）。

请看下面这个例子。假设光标停在如下所示的位置上：

```
(mary bob (dave (pete)) ed)
```

按下 “**ESC C-f**” 组合键时，光标将移动到如下所示的位置上：

```
(mary bob (dave (pete)) ed)
```

也就是说，光标将移过 S- 表达式 “(dave (pete))” —— 这一项本身是一个列表。但如果光标位置是下面这样的：

```
(mary bob (dave (pete)) ed)
```

按下 “**ESC C-f**” 组合键时，它就将移动到：

```
(mary bob (dave (pete)) ed)
```

第二个例子里的 S- 表达式是原子项 “bob”。

表 12-5 列出了对列表进行处理的有关命令。



表 12-5：列表处理命令速查表

键盘操作	命令名称	动作
ESC C-n	forward-list	移动到上一个列表
ESC C-p	backward-list	移动到下一个列表
ESC C-d	down-list	向前移动，进入下一级括号层次
(无)	up-list	向前移动，退出当前的括号层次
ESC C-u	backward-up-list	向后移动，退出当前的括号层次

为了便于记忆，可以把列表想像为文本行，把 S- 表达式想像为字符；因此，“C-n”和“C-p”出现在列表移动命令清单里，而“C-f”和“C-b”则出现在 S- 表达式移动命令清单里。“**ESC C-n**”和“**ESC C-p**”的工作情况，分别类似于“**ESC C-f**”和“**ESC C-b**”，但在光标的前或后必须有一个可以让光标移动经过的列表项。换句话说，光标位置上、光标后面或光标前面必须有一个左括号或右括号。如果没有括号，Emacs 就会报告出现一个错误。比如说，如果把光标放在如下所示的地方：

```
(fred bob (dave (pete)) ed)
```

并按下“**ESC C-n**”组合键，Emacs 就会给出如下所示的一条出错信息：

```
Containing expression ends prematurely
```

但如果把光标放在如下所示的位置上：

```
(fred bob (dave (pete)) ed)
```

则“下一个列表”其实就是“(dave (pete))”。如果在这个时候按下“**ESC C-n**”组合键，光标就会到达如下所示的位置：

```
(fred bob (dave (pete)) ed)
```

以列表项为单位进行移动的命令，使光标能够进入或者退出列表。比如，假设光标位于：

```
fred bob (dave (pete)) ed)
```

则按下“**ESC C-d**”组合键将使光标移动到：

```
(fred bob (dave (pete)) ed)
```

出现这种结果的原因是“**fred**”就是括号中列表项的下一级层次。再按下“**ESC C-d**”组合键将使光标移动到：

```
(fred bob (dave (pete)) ed)
```

现在进入到列表“(dave (pete))”里。此时，按下“**ESC C-u**”组合键将得到与“**ESC C-d**”组合键效果正好相反的结果，即让光标回退到这两级列表的外面。但如果输入的是“**ESC x up-list RETURN**”命令，就将向前越过这两级列表，得到如下所示的结果：

```
(fred bob (dave (pete)) ed)
```

表 12-6 对以函数定义 (defun) 为单位进行移动的命令进行了汇总，它们并不难理解。

表 12-6：以函数定义 (defun) 为单位进行移动的命令速查表

键盘操作	命令名称	动作
ESC C-a	beginning-of-defun	移动到当前函数的开头
ESC C-e	end-of-defun	移动到当前函数的结尾
ESC C-h	mark-defun	把光标放到函数的开头、把文本块标记放到函数的结尾

这些命令只有在当前函数的“(defun”关键字出现在某个代码行的最开始处时才能正常工作。

LISP 编辑模式提供了配对左括号的快速“闪现”功能；如果配对的左括号不在当前窗口的显示范围内，它所在的那一行语句将显示在辅助输入区里。LISP 编辑模式还为 TAB 键和 LINEFEED (“C-j”组合键) 提供了换行加缩进的自动功能（但 LISP 交互编辑模式不在此列，参见下面的内容）。

LISP 编辑模式所支持的缩进样式“知道”很多关于 LISP 关键字和列表语法方面的知识，但遗憾的是它并不那么容易定制（注 6）。

注 6： 缩进样式是由实现 Emacs LISP 模式的 LISP 代码决定的。如果你是一位有经验的 LISP 老手，就可以对 Emacs LISP 目录里的 **lisp-mode.el** 文件中的代码进行检查，并决定如何按自己的意愿对缩进样式进行定制。一个比较好的出发点是函数 **lisp-indent-line**。

请看下面的例子。这是本章前面内容中的 C 语言函数“times”的LISP写法，它展示了LISP在处理缩进样式方面的各种功能：

```
(defun times (x y)
  (let ((i 0)
        (result 0))
    (while (< i x)
      (setq result (+ result y)
            i (1+ i)))
    result))
```

基本的缩进量是 2：只要下一行上的代码又向下嵌套了一级，Emacs 就将使用这个值对它进行缩进。请看上面的例子，函数体（即从“defun”语句的下一行开始的代码块）向右缩进两个字符。而“(while . . . and result)”之间的语句又相对于“let”向右缩进了两个字符，这是因为它们是“let”引入的另外一个语句块。

虽然在行为上看起来像是关键字，可 defun、let 和 while 之类其实都是函数调用。函数调用所使用的缩进样式是这样的：如果函数有一个以上的参数，则该函数的函数名及其第一个参数，将出现在第一个代码行上，其他参数分别列在后续的代码行上，并且要与第一个参数对齐。换句话说，带有多个参数的函数应该写成下面这个样子：

```
(function-name arg1
                 arg2
                 arg3
                 ...)
```

刚才定义的 times 函数里，带有多个参数的 setq 就是一个很好的例子。

但语句“(result 0)”所在的那一行的缩进格式，却表明非函数调用的列表遵守的是不同的规则。问题中的列表其实是“((i 0) (result 0))”，即由两个元素（这两个元素都是列表）组成的一个列表，所以 LISP 编辑模式所支持的缩进样式将把这两个元素对齐摆放。

虽然类似关键字的 let 和 while 等术语实际上是函数调用，可 LISP 编辑模式却能根据自己对这类函数的“理解”对它们进行特殊的缩进安排。比如，如果我们打算把 while 循环的条件判断语句放到另外一行，并用 TAB 键对它进行准确的缩进，就会得到下面这样的结果：

```
(while
  (< i x)
  (setq result (+ result y)
        i (1+ i)))
```

类似的事情也会发生在 `if` 和 `cond` 等控制结构身上；第十三章里有一些很好的缩进示例。

缩进方面另外一个值得注意的事项是：LISP 编辑模式习惯于把多个右括号都连续地写在同一行的最末尾，而不是把它们分别写到不同的代码行上。比如，在上面的函数定义中，语句 “`i (1+i))`” 里就包含着“`1+`” 函数、`setq` 语句和 `while` 语句等 3 项内容的右括号。如果愿意，当然可以把这些右括号分别写到不同的代码行上；但如果用 **TAB** 键来对它们进行缩进，那么它们一般不会和与它们配对的左括号排列为一条直线——必须亲自动手来对它们进行缩进。

LISP 编辑模式不仅可以用 **TAB** 和 **LINEFEED** 命令进行缩进，还支持命令 “**ESC C-q**”（命令名是 `indent-sexp`），它的作用是对跟在光标后面的 S- 表达式的每一行进行适当的缩进。可以用这条命令对整个函数定义进行缩进——只要把光标放在 “`defun`” 的右面再按下 “**ESC C-q**” 组合键即可。

LISP 编辑模式里的注释由通用的注释命令 “**ESC ;**” 负责处理，它会按变量 `comment-column` 所设定的缩进量（或者，如果代码语句超出了这个位置，就缩进到语句最后一个字符再往右一个字符的位置）对代码中的注释进行缩进，插入一个分号 (;)，然后把光标放到分号的后面。如果想让注释占据整个一行（或者想从 `comment-column` 以外的其他任意位置开始写注释），就得亲自动手把光标移动到想让注释开始的地方并输入一个分号。如果注释内容占据了整个一行，并且又在这一行上按下了 **TAB** 键，则注释将被移动到 `comment-column` 所指定的位置。为了避免出现这种情况，可以用两个或者更多个分号来代替单个的分号。**TAB** 会把这样写的注释留在原处而不对它们进行缩进。LISP 编辑模式还支持在本章前面介绍的其他注释命令，比如 “**ESC J**” 命令（用来在下一行继续写注释）和 “**ESC x kill-comment RETURN**” 命令（清除一条单行的注释）等。以上介绍的这些功能适用于全部三种 LISP 编辑模式；接下来，我们将介绍它们各自所独有的功能。

Emacs LISP 模式是为运行于 Emacs 内部的 LISP 代码准备的；所以它支持直接运行输入的代码。因为 LISP 是一种解释型语言（与此相对的是纯粹的编译型语言），所以 LISP 程序设计过程中代码的书写和运行 / 调试阶段之间的界线是比较模糊的；

Emacs LISP 模式利用了 LISP 语言的这个特点，而稍后将要介绍的 LISP 交互模式就更胜一筹。在 Emacs LISP 模式里，命令 “**ESC C-x**”（命令名是 **eval-defun**）能够对光标周围或者光标后面的函数定义进行求值，也就是说，它将对该函数进行语法分析并把它保存起来，以便 Emacs 在用户调用这个函数的时候“知道”有这样一个函数。

Emacs LISP 模式还准备了一个 “**ESC TAB**” 命令（命令名是 **lisp-complete-symbol**），它的作用是自动补足光标前面的符号（比如变量、函数名等）。自动补足功能在第六章已经介绍过了。也就是说，如果输入某符号的前几个字符并按下 “**ESC TAB**” 组合键，Emacs 就会设法自动补足这个符号的名字。如果它能自动补足符号名，那么当然是继续工作；可如果它无法自动补足符号名，就说明刚才输入的前几个字符不能独一无二地确定一个符号名。需要再多输入几个字符（以进一步确定那个独一无二的符号）；或者，也可以再按一次 “**ESC TAB**” 组合键，屏幕上将弹出一个帮助窗口，里面是各种可供挑选的符号。接下来，可以多输入一些字符以亲自补足那个符号，或者再让 Emacs 去自动补足。

但 LISP 语言的处理工具并非只有 Emacs LISP 解释器一种，LISP 模式（相对于 Emacs LISP 模式而言）就是为使用其他 LISP 语言处理工具的用户而准备的。它提供了几个用来与外部 LISP 解释器进行通信的接口命令。LISP 模式里的 “**C-c C-z**” 命令（命令名是 **run-lisp**）将把用户系统的 LISP 解释器启动为一个子进程，并且会创建一个名为 “***lisp***” 的编辑缓冲区（以及一个与之关联的窗口）供输入和输出之用（注 7）。如果已经有一个 LISP 子进程存在，“**C-c C-z**” 命令就会使用它而不是另外再创建一个。如果把光标放在某个函数定义当中再按下 “**ESC C-x**” 组合键（此时它代表的是 **lisp-send-defun** 命令），就能把该函数定义送往 LISP 子进程中。这个过程使定义的函数在 LISP 解释器那里挂上号，今后就可以调用这些函数了。

如果编辑的文件内容全部都是 Emacs LISP 代码（例如正在设计编写自己的编辑模式（参见第十三章），或正在对某个现有的编辑模式进行修改），那么 Emacs LISP 模式是最佳的工具。但如果只是想编辑一点“小” LISP 代码——比如往 “**.emacs**” 文件里添加语句或者对它稍做修改，就可以选用 Emacs 作为另外准备的功能更强大的编辑功能，这些编辑功能将进一步模糊编写和运行代码两项工作之间的界线。

注 7： LISP 编辑模式的这条命令（**run-lisp**）是为 BSD UNIX 系统上的 franz LISP 系统准备的，但它也能用来启动其他的 LISP 解释器。

这些更强大的功能中的第一个是“**ESC :**”命令（命令名是**eval-expression**）。这个命令可以在辅助输入区里输入任何一种形式的单行LISP表达式，然后对该表达式进行求值，再把其结果显示在辅助输入区里。这种方法特别适合用来检查Emacs变量的值，和试用那些没有按键绑定，或需要额外参数的Emacs“内部”函数。在使用**eval-expression**命令的时候，还可以利用符号自动补足命令“**ESC TAB**”。

不幸的是（或者说幸运的是，这取决于不同的观点），Emacs在一般情况下不可以使用**eval-expression**命令。如果试着按“**ESC :**”组合键，就会在辅助输入区里看到一条“*loading novice . . .*”消息，接着会弹出一个窗口询问“你并不是真的想输入这条命令，对吗？”这条消息。它下有3项选择：按空格键只执行这条命令一次；“**y**”执行这条命令并激活它以便今后继续使用，而且今后也不会再询问刚才的问题；“**n**”则代表什么也不做。

如果真的想使用**eval-expression**命令，请回答“**y**”。这个命令的执行效果是把下面这条语句添加到“*.emacs*”文件里：

```
(put 'eval-expression 'disabled nil)
```

如果你是一位老练的LISP程序员，就会知道添加到“*.emacs*”文件里去的这条语句将把符号“**eval-expression**”的“**disabled**”属性设置为“**nil**”。换句话说，Emacs认为某些命令是不适合初级用户使用的，因此它会在默认的情况下禁用这些命令。如果你想跳过这些步骤而直接使用**eval-expression**命令，只要简单地把上面这条语句放到“*.emacs*”文件里去即可（那可别忘了输入几个单引号）。

能够帮助用户熟悉Emacs LISP代码的命令是“**C-x C-e**”（命令名是**eval-last-sexp**）。这个命令的作用是运行光标所在的代码行上的那条LISP语句，然后把其执行结果显示在辅助输入区里。“**C-x C-e**”非常适合用来测试Emacs LISP文件里的单条语句。

LISP交互模式的功能就更强大了。这是“***scratch***”编辑缓冲区默认进入的编辑模式。不带后缀名的文件名一般都会让Emacs进入LISP交互模式（可以用修改**auto-mode-alist**变量的方法来改变Emacs的这种默认行为），有关操作步骤请参考本章前面的内容，更详细的讨论见第十一章。也可以用“**ESC x lisp-interaction-mode RETURN**”把任意一个编辑缓冲区放到LISP交互模式里；如果想另外创建一个处于LISP交互模式的编辑缓冲区，可以按下“**C-x b**”组合键（命令名是**switch-to-buffer**），提供一个编辑缓冲区名字，最后把它放到LISP交互模式下即可。

除了把 **LINEFEED** (即“C-j”组合键) 绑定为 **eval-print-last-sexp** 命令之外, LISP 交互模式与 Emacs LISP 模式可以说是一模一样。这个命令的作用是对光标前面的 S-表达式进行求值, 然后把其结果显示在该 S-表达式所在的编辑缓冲区里。要想获得其他编辑模式绑定在 **LINEFEED** 上的换行及缩进的 **newline-and-indent** 命令效果, 必须先按回车键再按 **TAB** 键。

请记住, LISP 语言里的 S-表达式可以是任何一个语法正确的表达式。因此, 可以在 LISP 交互模式里利用 **LINEFEED** 检查变量的值、输入函数定义和运行函数等等。举例来说, 如果在输入变量 **auto-save-interval** 后按下了 **LINEFEED**, 就能查看到这个变量的值 (默认值是 300)。如果输入一个函数定义后在它最后一个右括号的后面按下了 **LINEFEED**, Emacs 就会把这个函数保存起来 (以便今后调用) 并打印出它的名字; 就这种情况而言, **LINEFEED** 的作用类似于 “**ESC C-x**” 组合键 (命令名是 **eval-defun**), 只不过它要求光标必须位于定义的函数的后面 (而不是放在函数定义的前面或中间)。如果输入的代码是对某个函数的调用, Emacs 就会求值 (运行) 这个表达式并把该函数的返回值都显示出来。

LISP 交互模式中的 **LINEFEED** 是试用、渐进式开发和调试 LISP 代码的绝佳手段。同时, 因为 Emacs LISP 本身就是“真正的” LISP 语言, 所以甚至可以用它来为其他的 LISP 系统开发一些代码。

FORTRAN 模式

当访问后缀名是 “*f*” 或 “*for*” 的文件时, Emacs 就会自动进入 FORTRAN 模式。FORTRAN 模式有几项功能专门用来处理这种面向字符列位置的格式。它不仅能识别和理解其他语言编辑模式所共有的语言学元素, 还支持一些专门用来对语句和子程序进行处理的命令; 我们把这些命令列在表 12-7 里。

表 12-7: FORTRAN 模式中的移动命令速查表

键盘操作	命令名称	动作
C-c C-n	fortran-next-statement	向前移动一个语句
C-c C-p	fortran-previous-statement	向后移动一个语句
ESC C-a	beginning-of-fortran-subprogram	移动到当前子程序的开头

表 12-7: FORTRAN 模式中的移动命令速查表 (续)

键盘操作	命令名称	动作
ESC C-e	end-of-fortran-subprogram	移动到当前子程序的结尾
ESC C-h	mark-fortran-subprogram	把光标放到子程序的开头，把文本块标记放到子程序的结尾

“C-c C-n”和“C-c C-p”命令与普通的“C-n”和“C-p”命令是不一样的，它们能跳过程序中的续行和注释行。

FORTRAN 模式也支持其他语言编辑模式所支持的缩进和注释命令，但在做法上有一些我们稍后就会介绍给大家的变化。除通用的注释命令“ESC ;”外，FORTRAN 模式还准备了一个对包围光标的子程序进行适当缩进的“**ESC C-q**”命令（命令名是**fortran-indent-subprogram**），和一个把光标和文本块标记之间的区域里的语句全部设置为注释行的“**C-c ;**”命令（命令名是**fortran-comment-region**）。

FORTRAN 模式在缩进、续行和注释样式等问题上的每一个侧面几乎都能通过相应的 Emacs 变量来进行定制（注 8）。我们把这些变量列在表 12-8 里，然后给出它们的几个用法示例。

表 12-8: FORTRAN 模式中的 Emacs 变量

变量	默认值	说明
fortran-minimum-statement-indent	6	开始输入语句之前需要保留的缩进量（例如：缩进量为 6 时语句将从第 7 列开始输入）
fortran-do-indent	3	do 语句块里的语句还需要增加的缩进量
fortran-if-indent	3	if 语句块里的语句还需要增加的缩进量
fortran-continuation-char	\$	续行时放在第 6 列里的续行标记字符
fortran-continuation-indent	5	语句续行时的缩进量（比如从第 6 列开始再缩进 5 个字符的位置）

注 8：事实上，甚至可以把 FORTRAN 模式定制得让编译器认不出那是 FORTRAN 代码来！

表 12-8 FORTRAN 模式中的 Emacs 变量 (续)

变量	默认值	说明
fortran-comment-line-column	6	占据一行的注释所使用的缩进量; “ ESC ; ” 命令使用的就是这个值
fortran-comment-indent-style	“fixed” (固定值)	这个变量的值可以取 “ nil ”、“ fixed ” 或 “ relative (相对值)”, 注意后两个值的前面带有单引号; 请参见下面的用法示例
comment-start	“nil”	与程序代码同在一行上的注释的前导字符; 请参见下面的用法示例。如果这个变量取值为 “ nil ”, 则表示不允许这样的注释
fortran-line-number-indent	1	行号的最大缩进量, 用来保证行号不会延伸到第 5 列 (续行标记列) 上。请参考后面对语句自动编号功能的讨论
fortran-comment-region	“ c\$\$\$ ”	由 fortran-comment-region (“C-c ;”) 命令插入到每一行前面去的字符串

下面用几个具体的例子来演示一下这些功能的用法。假设用FORTRAN语言写出一个“times”函数, 而它的功能与本章前面给出的那个同名的C语言函数是一样的:

```
integer function times (i, j)
times = 0
do 10 k = 1, i
      times = times + j
10  continue
      return
      end
```

输入这些代码行的时候, 按下 **TAB** 键会把光标先移动到第 7 列——也就是开始输入语句的地方 (这个缩进量由 **fortran-minimum-statement-indent** 变量控制)。如果输入的是 **do** 或 **if** 语句, Emacs 还将根据变量 **fortran-do-indent** 和 **fortran-if-indent** 的取值情况再对它们做相应的缩进 (注 9)。请注意, 输入 “**continue**” 的时候, 它先会像 **do** 循环的循环体语句那样被缩进, 所以必须再按下 **TAB** 键才能让

注 9: FORTRAN 模式的这项功能只有在关键字 **do** 或 **if** 之后, 至少有一个空格的情况下才能发挥作用, 可大多数编译器却没有这样的要求。

Emacs 调整其缩进量以对齐与之配对的“do”。用来结束 if 循环的“continue”语句也需要做同样的处理。(FORTRAN 模式还能识别和理解用来结束 do 循环的“end do”和“enddo”以及用来结束 if 循环的“else”、“elseif”、“endif”和“end if”。) 另一个必须注意的地方是语句块不能共用同一个“continue”语句，每个 if 或 do 都必须有它自己的“continue”。因为只有这样，FORTRAN 模式的自动缩进功能才能正确工作。

FORTRAN 模式提供了一个名为“自动编行号 (electric line numbering)”的功能来处理FORTRAN程序里的行号问题。如果在开始输入某行代码之前先输入了一个数字，FORTRAN 模式就将认为输入的是一个行号，而它就将据此对这个数字以及后续输入的其他数字进行正确的缩进——它会把这个数字挪到第 2 列（这个缩进量由变量 **fortran-line-number-indent** 控制）上，然后再把光标放到第 7 列上，以便直接开始输入这一行上的FORTRAN语句。后续输入的数字将“加入”到行号里，而光标仍将返回到第 7 列。

请看下面这个用法示例。假设想给代码行加上一个行号“10”；光标现在位于这一行的最开始处。于是，输入数字“1”的时候，结果将是这样的：

1



接着输入数字“0”的时候，它会和数字“1”结合在一起构成行号“10”，如下所示：

10



注意光标会回到第7列等待输入这一行的语句。最多可以输入一个5位数的行号；输入行号的第5位数字时，行号的缩进量将被调整为1以留出必要的空间，如下所示：

12345



如果试图输入一个超过 5 位数字的行号，FORTRAN 模式将在辅助输入区里显示一条警告信息。

FORTRAN 模式支持的注释样式有两种：一种是标准样式，它要求注释必须自成一行，而且必须在第 1 列里加上一个字母“c”作为标志；另一种样式允许注释与语句出现在同一行上，语句后面的惊叹号字符“!”表示随后的文字内容是一条注释。值得注意的是很多编译器并不支持后一种风格的样式。在默认的情况下，FORTRAN

模式也不允许使用后一种注释样式，但可以通过把变量 **comment-start** 设置为惊叹号字符（“!”）的方法来启用它。

如果已经把 **comment-start** 设置为惊叹号字符（“!”）了，那么，当用“**ESC ;**”命令准备在已经输入有语句的代码行上输入注释的时候，Emacs 将根据变量 **comment-column** 的取值对它进行缩进（或者，如果那个位置已经有了文本，就缩进到该文本最后一个字符再往右一个格的位置），再插入一个惊叹号字符（!），然后把光标放在惊叹号的后面。此外，如果用“**ESC j**”组合键把注释延伸到下一行，Emacs 就将插入一个新行，缩进到变量 **comment-column** 设定的位置，再插入另外一个惊叹号字符（!）。如果变量 **comment-start** 设置为“**nil**”，那么，当用“**ESC ;**”命令准备在已经输入有语句的代码行上输入注释的时候，Emacs 就会在当前行的上面插入一个新行，然后在新行的第 1 列放上一个字母“c”——也就是插入一个整行的注释。

可以用变量 **fortran-comment-indent-style** 来控制整行注释的缩进样式。在一般情况下，如果在一个空白行上按下“**ESC ;**”组合键，Emacs 就将自动插入一个字母“c”并缩进到第 7 列去。如果试图在第 7 列之前开始输入注释文本，那么 Emacs 会在按下 **TAB** 键的时候（或者在用“**ESC C-q**”组合键对整个子程序进行缩进的时候）把它重新缩进到第 7 列。这种行为是由变量 **fortran-comment-indent-style** 的默认值设定的，这个默认值是“**'fixed**”（注意它前面的单引号）。

不过，也许有的编译器允许注释文本占据第 7 列之前的位置。如果想利用这一点，就需要把变量 **fortran-comment-indent-style** 设置为“**nil**”。此后，就可以在字母“c”标志后面的任意位开始输入注释文本；而且，即使按下了 **TAB** 键或“**ESC C-q**”组合键，Emacs 也不会改变缩进设置。另外一种选择是：如果想让注释文本的缩进量超过 7 个字符，就需要把变量 **fortran-comment-indent-style** 设置为“**'relative**”（注意它前面的单引号），这将把注释文本的缩进量设置为 7 加上变量 **fortran-minimum-statement-indent** 的值；也就是从第 7 列开始再多缩进 6 列（以变量的默认值为例），总的缩进量将达到 13 列。

FORTRAN 模式的另一个特色是它的缩略词汇（参见第三章）表，这里所说的缩略词汇是 FORTRAN 语言中的关键字的缩写，可以利用它们加快代码的输入速度。FORTRAN 模式的缩略词汇表其实是利用分号（;）字符对 Emacs 原有的简写词功能进行的扩展。为了避免与设置的其他简写词发生冲突，FORTRAN 模式的缩略词

汇全都以分号(;)打头;如果输入“;?”这两个字符,Emacs就会把全体FORTRAN缩略词汇列成一份清单显示在一个“*Help*”窗口里。

要想在FORTRAN模式里使用缩略词汇，需要先用“**ESC x abbrev-mode RETURN**”命令启用Emacs的简写词功能。然后，就可以用输入一个简写词再按空格键的方法来快速输入。每一个FORTRAN关键字都有相应的缩略形式；表12-9里列出的只是其中的一部分。

表 12-9: FORTRAN 模式下的一些缩略词汇

缩略词汇	关键字
;c	continue
;dp	double precision
;dw	do while
;f	format
;fu	function
;g	goto
;in	integer
;p	print
;rt	return
;su	subroutine

我们来看几个例子。如果想定义一个函数（并且已经在简写模式里），则直接输入“;fu”再按下空格键即可。如果输入“;fu”之后停了下来，Emacs会把刚输入的字符串显示在辅助输入区里以提醒用户当前正在使用一个缩略词汇；按下空格键之后，辅助输入区里的“;fu”字样就会消失，取而代之的是关键字“function”。

最后再介绍几个用在 FORTRAN 模式里的杂项命令。“**ESC C-j**”组合键（命令名是 **fortran-split-line**）所做的事正好与“**ESC ^**”（把这一行和上一行合并为一行）相反：它把当前行从光标位置一分为二，给第二行加上一个续行标记字符，再让它多缩进 6 列。“**C-c C-r**”组合键（命令名是 **fortran-column-ruler**）在当前行的上方显示一把由两行字符组成的标尺，如下所示：



标尺是临时性的，它会在按下任意一个键之后消失。最后，“**C-c C-w**”组合键（命令名是 **fortran-window-create**）会把当前窗口在水平方向上收缩为 72 列宽，这可以帮助控制语句不至于太长。标尺命令可以帮助完成用“**ESC C-j**”组合键分断当前行的操作。

对程序进行编译

我们在本章的开头已经讲过，Emacs 向程序员提供的支持并不仅限于写代码的时候。当工作在大型程序设计项目上的时候，一个典型的 Emacs 使用策略是登录上机、进入存放有源代码文件的目录、启动 Emacs 来编辑所有的源代码文件（比如 C 程序员使用的“**emacs Makefile *.[ch]**”命令）。在对源代码进行编辑的过程中、可以用下面将要介绍的命令对它进行编译（正如大家将要看到的那样），根本不用考虑保存修改。还可以利用 Emacs 的 shell 模式（参见第五章）在一个 shell 窗口里对经过编译的代码进行测试。总之，就整个上机过程而言，你几乎完全不用退出 Emacs。

Emacs 准备了一个面向编译器和 UNIX 的 **make** 工具的接口；与 shell 模式相比，这个接口更直接、更强大。这个接口的核心是命令“**ESC x compile RETURN**”，它将引发一连串的事件。

首先，它会提示输入一个编译命令。默认的编译命令是“**make -k**”（注 10），但如果输入了另外一个命令，新的命令就将在本次 Emacs 会话过程中以后的编译操作里取代原来的默认编译命令。如果想改变默认的编译命令，可以在“**.emacs**”文件里对变量 **compile-command** 进行设置。

输入编译命令之后，Emacs 会自动把尚未存盘的编辑缓冲区保存起来；而保存尚未存盘的修改本来是个人的责任，Emacs 会在这方面减轻用户的负担。接着，它在一个关联窗口里创建一个名为“***compilation***”的编辑缓冲区。然后，它运行编译命令（编译命令是作为一个子进程来运行的，就像 shell 模式里的 shell 一样），编译器的输出显示在“***compilation***”编辑缓冲区里。在编译命令执行期间，在状态行上显示“Compiling: run”字样；编译作业结束时，则显示“exit”字样。

注 10：“-k”选项的作用是取消 **make** 在某个作业返回一个错误之后就停止执行的默认行为。加上这个选项之后，即使依赖树（dependency tree）的某个分支出了错，**make** 也不会停止执行，它会继续对不依赖于该出错分支的其他分支进行处理。

下面的事情就更有意思了。如果编译结果报告说代码里有错误，那么可以用“**C-x `**”组合键（命令名是 **next-error**，注意第二个字符是一个反引号而不是一个单引号）来检查代码：Emacs 会读入第一条出错信息，把出现错误的文件和行号找出来，然后把光标移到该文件和该行号指定的地方。修改完错误之后，如果再次按下“**C-x `**”组合键，就会来到下一个出现错误的地方。每按一次“**C-x `**”组合键，Emacs 就会卷动“*compilation*”窗口里的内容，让当前的出错信息显示在该窗口的顶部。

如果想从第一条出错信息处重新开始，就需要给“**C-x `**”加上一个前缀（即按下“**C-u C-x `**”组合键）。“**C-x `**”组合键还有这样一个好处：可以在看到出错信息之后立刻按下它，用不着等到整个编译工作都结束之后再使用它。

“*compilation*”编辑缓冲区的编辑模式（即 Emacs 的编译模式）还支持其他一些用来浏览出错信息的命令；我们把这些命令汇总在表 12-10 里。

表 12-10：编译模式命令速查表

键盘操作	命令名称	动作
C-x `	next-error	移动到下一条出错信息并访问与之对应的源代码
ESC n	compilation-next-error	移动到下一条出错信息
ESC p	compilation-previous-error	移动到上一条出错信息
C-c C-c	compilation-goto-error	访问对应于当前出错信息的源代码
SPACE	scroll-down	下卷屏幕显示内容
DEL	scroll-up	上卷屏幕显示内容

SPACE 键和 **DEL** 键是各种 Emacs 的“只读”模式——比如 RMAIL 模式（参见第六章）所使用的卷屏命令，它们用起来很方便。

请注意，“**ESC n**” 和 “**ESC p**” 并不是用来访问对应于出错信息的源代码的；它们的操作对象是出错信息本身。如果出错信息的内容多于一行，用它们来移动会比较方便些。如果想访问对应于某条出错信息的源代码，就需要使用 “**C-c C-c**” 组合键。

那么，Emacs 又是如何对出错信息进行分析的呢？它使用变量 **compilation-error-regexp-alist** 对出错信息进行分析；这个变量是一个由正则表达式构成的列表，它的作用是对很多种 C / C++ 编译器以及 C 代码检查程序 **lint** 所给出的出错信息进行匹配（注 11）。这个变量与 Emacs 其他语言编辑模式所对应的程序设计语言（比如 FORTRAN、Ada 和 Modula-2 等）的编译器也能配合工作。Emacs 会用该列表里的每一条正则表达式对一条出错信息进行尝试性分析，直到它找到那条能够用来准确地提取出错误发生地点的文件名和行号的正则表达式为止。

尽管已经尽了很大的努力，Emacs 的出错信息分析器无法与某些编译器配合工作的情况还是有可能发生；尤其是在一个非 UNIX 系统上使用 Emacs 的时候更容易出现这种情况。拿一些你知道肯定包含有某些特定错误的程序代码来试试 “**ESC x compile**” 命令就可以做出判断：如果 Emacs 在按下 “**C-x `**” 组合键时报告说 “no more errors (没有其他错误)”，就说明 **next-error** 命令不能与编译器配合工作。

如果 Emacs 的出错信息分析器不能与编译器配合使用，则可以采取下面这种方法来解决这一问题：给变量 **compilation-error-regexp-alist** 增加一条符合编译器的出错信息格式的正则表达式。我们将在第十三章内容里介绍一个具体的例子。

compile 程序包还包括有为 UNIX 的 **grep**（查找文件）命令提供的类似支持，这就使 Emacs 具备了在多个文件里进行查找的能力。如果输入 “**ESC x grep**” 命令，Emacs 就会提示输入一些准备传递给 **grep** 的参数——即一个查找模板（search pattern）和一组文件名。然后，Emacs 会调用执行带 “**-n**” 选项的 **grep** 命令，这个选项的作用是告诉 **grep** 把匹配到的代码行的文件名和行号显示出来（注 12）。以后的事情就和 “**ESC x compile**” 命令的后续操作情况一样了；可以用按下 “**C-x `**” 组合键的方法让 Emacs 访问 **grep** 所查找的文件里的下一个匹配行。

注 11：令人遗憾的是，Emacs 无法对关于 **make** 自身的出错信息进行分析——比如制作文件（**makefile**）中的语法错误。

注 12：如果只用 “**grep -n**” 命令对一个文件进行操作，它就将只给出行号；但 Emacs 会通过给 **grep** 命令增加一个哑文件参数 **/dev/null** 的办法，来强制它在这种情况下也把文件名显示出来。

第十三章

用 LISP 语言对 Emacs 做 进一步开发

本章内容：

- LISP 语言简介
- LISP 语言的基础函数
- Emacs 的内部函数
- 主编辑模式程序设计实例
- 对现有编辑模式进行定制
- 建立自己的 LISP 开发库

Emacs 并不是万能的。如果你使用 Emacs 已经有了一段时间并且用过它的一些高级功能，就肯定会发现一些 Emacs 无法实现的功能。虽然 Emacs 有好几百条内部命令、好几个程序包和编辑模式以及许多诸如此类的东西，但终将会出现这样一种情况：你需要的某些功能是 Emacs 不具备的——或者说至少在缺省情况下是如此。不论你发现它缺少的是什么样的功能，都可以自己动手，用 Emacs LISP 程序实现它。

但在进入正题之前，我们想提醒大家一句：并非每个人都适合学习这一章内容。它是为那些已经能熟练使用 Emacs 的人们和那些有程序设计经验（并不非得是 LISP 程序设计经验）的人们准备的。如果你没有这样的经验，那最好还是跳过本章；如果你就是想用 Emacs 来完成某些事情，那么可以去找个友好的 Emacs LISP 老手来帮忙，让他替你写出必要的程序代码。

再者，我们并不打算在这一章里对 LISP 程序设计做全面细致的讨论，那是另外一本书的事。我们的侧重点在于介绍这种语言的基本概念和编写 Emacs 代码时常用的命令和技巧。如果你想在本章的基础上做进一步的学习，可以去研读自由软件基金会（他们的地址见附录一）提供的《Gnu Emacs LISP Reference Manual》(GNU Emacs LISP 参考手册)；或者任何一本 LISP 教科书（Winston 和 Horn 合著的《LISP》是一个不错的选择；这本书是由 Addison-Wesley 出版社出版的）。

Emacs LISP 是一种全功能的 LISP 实现（注 1）；因此，与大多数文本编辑器所普遍使用的宏（macro）或脚本语言相比，它提供了更强大的功能。（本书作者之一曾经完全依靠 Emacs LISP 写出过一个小型的专家系统。）事实上，完全可以把 Emacs 看做是一个带有很多内部函数的 LISP 系统，而其中许多函数是用来实现文本处理、窗口管理、文件 I/O 和其他各种与文本编辑有关的功能的。Emacs 的源代码是用 C 语言写的。它只实现了 LISP 解释器、LISP 指令和一些最基本的文本编辑命令；Emacs 的其他功能则是在它们的基础上用一个规模巨大的内部 LISP 代码层实现的。Emacs 的最新版本大约带有 200,000 行 LISP 代码。

LISP 语言在许多方面与 C 或 Pascal 等其他常用的程序设计语言是相似或者相通的，这一章就将从程序设计语言的共通之处开始讲起。但这些基础性的功能已经足以让人们写出许多 Emacs 命令了。接下来，我们再一起去探讨怎样才能把 LISP 代码与 Emacs 衔接起来，让编写的 LISP 函数能够成为 Emacs 的编辑命令。我们会见到许多能帮助写出 Emacs 命令的 LISP 内部函数，其中有不少都用到了正则表达式；我们已经在第三章里对正则表达式做过简单的介绍，在此基础上，我们将在本章以 LISP 程序设计为侧重点对它做更进一步的探讨。在此之后，我们将回到 LISP 的基本概念上，把这种语言在列表（list）处理方面的独到之处介绍给大家；然后，我们将通过实例向大家演示一下怎样才能写出一个主编辑模式的程序代码，虽然这个主编辑模式很简单，但本书各章所介绍的概念都将集中体现在里面。然后，我们将介绍如何在不改动（甚至用不着查看）用来实现 Emacs 内部的各种主编辑模式的源代码的前提下，对它们进行定制，你们将会看到这有多么简单。在本章的结尾，我们将介绍如何建立自己的 LISP 程序包。

LISP 语言简介

大家可能听说过 LISP 是一种用在人工智能（artificial intelligence, AI）方面的程序设计语言。即使你不了解人工智能，也没有什么好担心的。LISP 语言的语法可能不同寻常，但它的基本功能与 C 或 Pascal 等较为常见的程序设计语言却没什么不同；而这一章的学习重点正是这些基本的功能。在介绍完 LISP 语言的基本概念之后，我

注 1： 有经验的 LISP 程序员会发现 Emacs LISP 最接近 MacLISP，但增加了一些 Common LISP 的功能。更完整的 Common LISP 仿真效果可以通过加载 cl 扩展包实现（参见附录四）。

们将编写出各种示范性的函数，而你完全可以把它们用到 Emacs 里。为了能够试用这些示范性的函数，你需要熟练掌握 Emacs 中的 Emacs LISP 模式和 LISP 交互模式，而这两种编辑模式都已经在第十二章介绍过。

LISP 语言的基本元素

函数、变量和原子项是 LISP 语言最基本的元素，也是每一个想要学习 LISP 语言的人所必须熟悉和掌握的概念。函数是 LISP 语言中惟一的程序单元；其地位相当于其他程序设计语言中的各种过程（procedures）、子例程（subroutines）、程序（programs）甚至是操作符（operators）。

LISP 语言中的“函数”定义为由上述基本元素组成的各种“列表”（list），而且这种列表往往还嵌套调用着其他现有的函数。一切函数都有“返回值”（就像 C 或 Pascal 语言中的函数一样）；一个函数的返回值就是该函数最后一个列表项的值，它通常就是最后一个函数调用所返回来的值。嵌套在另外一个函数里的某个函数调用相当于其他程序设计语言中的一条“语句（statement）”，而我们将在本章内容里不加区别地使用“语句”和“函数调用”进行讨论。函数调用的语法是：

```
(function-name argument1 argument2 ... )
```

它相当于 C 或 Pascal 语言中的：

```
function_name (argument1, argument2, ... )
```

这条语法适用于一切函数，那些相当于其他程序设计语言中的算术或比较操作符的函数也不例外。比如说，如果想对 2 和 4 做加法，在 C 或 Pascal 语言中要使用表达式“ $2 + 4$ ”；但在 LISP 语言里，必须把它写为：

```
(+ 2 4)
```

类似地，如果想表达“ $4 \geq 2$ ”（大于或等于），在 LISP 语言里就必须把它写为：

```
(>= 4 2)
```

LISP 语言中的“变量”与其他程序设计语言中的变量概念很相似，但 LISP 变量没有类型（type）的说法。一个 LISP 变量可以取任意类型的值。

“原子项”是任意类型的值，它可以是整数、浮点数（即实数）、字符串、布尔真/假值、符号，也可以是编辑缓冲区、窗口、进程等特殊的 Emacs 类型。下面是与这些原子项有关的语法规规定：

- **整数**——取值范围在 -2^{37} 到 $2^{37}-1$ 范围间的所有数字。
- **浮点数**——即数学上的实数，它们可以用小数点和科学计数法（用小写字母“e”来表示10的幂次）来表示。比如说，数值5489可以被表示为5489、5.489e3、548.9e1 等等。
- **字符**——LISP 语言中的字符必须带有一个前导的问号。比如像“?a”这样。**ESC**、**LINEFED** 和 **TAB** 分别用“\e”、“\n”和“\t”来表示；其他控制字符要加上前缀“\C-”。比如说，“C-a”字符表示为“?\C-a”（注 2）。
- **字符串**——必须放在一对双引号中间；字符串中的引号和反斜线字符必须用一个反斜线字符来引导。比如说，“Jane said, \"See Dick run.\"”是一个合法的字符串。
- **布尔值**——“t”表示真，“nil”表示假；在大多数需要使用一个布尔值的场合，任何一个不是“nil”的值都意味着真。“nil”在很多场合还用来表示一个空值（null）或无取值，我们马上就能看到具体的例子。
- **符号**——即 LISP 语言中各种事物的名字，比如变量或者函数的名字。有时候，我们需要使用的是事物的名字而不是它的值，这就需要在该事物的名字前面加上一个单引号（'）。举个例子，我们在第十一章介绍的**define-key** 函数使用的就是命令的名字（做为一个符号）而不是该命令本身。

一个能够集中体现这些 LISP 基本概念的简单例子是函数 **setq**（注 3）。大家可能已经从前面各章里总结出来，即 **setq** 是对变量进行赋值的一个方法，比如下面这条语句：

```
(setq auto-save-interval 800)
```

请注意，**setq** 是一个函数，这与其他程序设计语言使用“=”或“:=”之类的特殊

注 2：字符也可以用整数来表示。此时使用的是 ASCII 代码（在大多数机器上）。

注 3：我们希望 LISP 的卫道士们能够原谅我们把 **setq** 称为一个函数。我们这样做是为了能够不拘形式地把问题讲清楚，而这也正是它在技术方面的意义所在。

语法进行赋值的做法是不一样的。**setq** 要使用两个参数：一个变量名和一个值。在上面的这个例子里，变量**auto-save-interval**（两次自动保存操作之间等待的按键次数）的值被设置为 800。

setq 可以一次对多个变量进行赋值，请看下面这个例子：

```
(setq thisvar thisvalue  
      thatvar thatvalue  
      theothervar theothervalue)
```

setq 的返回值就是最后一个赋值，对上例而言就是“theothervalue”。对变量进行赋值还可以使用其他方法，但 **setq** 是适用面最宽的。

函数的定义

我们一起来看一个简单的函数定义示例。如果你使用的是终端，则可能想不带任何参数地启动 Emacs；这将使你进入编辑缓冲区“`*scratch*`”，它是一个 LISP 交互模式（参见第十二章）下的空白编辑缓冲区，你可以在这个编辑缓冲区里试用我们在本章给出的程序示例。

在介绍程序示例之前，大家有必要再多了解一些 LISP 的语法规定。首先，短划线字符（`-`）在变量名、函数名等语言元素里用做“间隔”字符。这是 LISP 程序设计中一种广泛使用的记号写法；它与 C 语言和 Ada 语言中下划线字符（`_`）的作用是一样的。另外一个更值得重视的问题是 LISP 程序代码中的括号。LISP 是一种“古老”的程序设计语言，在它诞生的年代里，人们对程序设计语言的语法问题还很少做细致的考虑，因此它的语法对程序员并不是非常友好。而用 LISP 语言写出的程序会使用大量的列表——也就是会大量地使用括号；从程序设计的角度看，这种做法有它的优点，大家学到这一章的末尾就会看到。

但从程序员的角度看，如何一个不少地把那么多的括号都正确地配上对却是个令人头疼的问题。使这一问题更加复杂的是，人们习惯于把多个右括号集中放置到代码行的末尾，而不是把它们分别放在一行并与对应的左括号对齐——后者的可读性当然要好得多。Emacs LISP 模式提供的支持功能（特别是它的 TAB 键缩进功能和配对括号“闪现”功能）为我们准备了最好的解决手段。

现在，来看看我们的示例函数。假设你是一名学生或记者，需要把一篇论文或者稿

件中的单词个数统计出来。Emacs 没有提供统计编辑缓冲区中单词个数的内部函数，所以我们打算写一个 LISP 函数来完成这项工作：

```
1. (defun count-words-buffer ()  
2.   (let ((count 0))  
3.     (goto-char (point-min))  
4.     (while (< (point) (point-max))  
5.       (forward-word 1))  
6.     (setq count (1+ count)))  
7.   (message "buffer contains %d words." count))
```

我们来对这个函数一行一行地进行分析，看它到底做了些什么。（当然，如果你是在一个终端上试用这个函数的，可别把行号也输入进去。）

第1行上的 **defun** 定义了一个函数的名字和它的参数。别忘记 **defun** 本身也是一个函数——它会在自己被调用时定义一个新的函数。（**defun**的返回值是一个符号，这个符号就是它所定义的函数的名字。）函数的参数都放在括号里，形成了一个由变量名组成的列表。这个例子里的函数没有参数。如果在某个参数的前面加上一个关键字“**&optional**”，就可以把这个参数设置为可选参数。如果一个参数是可选的并且没有在函数调用的时候给出来，它的取值就将是“**nil**”。

第2行上是一个 **let** 结构，这个结构的基本格式是：

```
(let ((var1 value1) (var2 value2) . . .)  
    statement-block)
```

let 做的第一件事是对变量 var1、var2 等进行定义，把它们的初始值设为 value1、value2 等等。接着 **let** 将执行它的语句块部分，语句块是顺序排列的一组函数调用或者值，就像函数的函数体一样。

为了便于理解，我们可以认为 **let** 做了 3 件事情：

- 定义（或者叫“声明”）了一个由变量组成的列表。
- 给变量设置初始值，就像 **setq** 那样。
- 创建一个“知道”那些变量的语句块；**let** 语句块也叫做变量的“作用域（scope）”。

对一个变量来说，如果它是用 **let** 定义的，就允许以后在 **let** 语句块里用 **setq** 对它的



值重新进行设置。再深入一些，可以给用 **let** 定义的变量起一个与某个全局变量相同的名字；在该 **let** 语句块里对该变量进行的一切 **setq** 操作都只作用于那个局部变量，那个同名字的全局变量不会受到任何影响。但如果用 **setq** 赋值的变量不是用 **let** 定义的，该变量就将被看做一个全局变量。要尽可能地避免使用全局变量的名字做为局部变量的名字，因为这很有可能会引起局部变量与同名的全局变量发生冲突。

回到我们的示例函数上来。我们用 **let** 定义了一个局部变量 **count**，并把它的初始值设为 0。我们马上就会看到，这个变量将用做循环计数器。

第 3 行到第 7 行是 **let** 语句块里的语句。第一条语句（第 3 行）调用了 Emacs 的内部函数 **goto-char**，这个函数我们最早是在第二章里看到的。**goto-char** 函数的参数是一个（嵌套着的）函数调用，被调用的函数也是 Emacs 的一个内部函数，它的名字是 **point-min**。我们在前面已经讲过，“**point**（插入点）”是 Emacs 内部对光标位置的称呼，我们将在本章以后的内容里有时会用“插入点”来称呼“光标”。**point-min** 的返回值是当前编辑缓冲区中第一个字符的位置，它的取值几乎总是 1；于是，**goto-char** 以数值 1 为参数被调用，整个语句的执行效果是把光标移动到编辑缓冲区的开头。

第 4 行安排了一个 **while** 循环，它与 C 语言或 Pascal 语言中的 **while** 循环没什么两样。LISP 语言中的 **while** 结构的基本格式是这样的：

```
(while condition  
      statement-block)
```

与 **let** 的情况相类似，**while** 也设置了一个语句块。循环条件 **condition** 是一个值（一个原子项、一个变量或者是有一个返回值的某个函数）。**while** 将对这个值进行测试；如果它是 “nil”，循环条件将被认为是真的，这个 **while** 循环也就到此为止。如果这个值不是 “nil”，循环条件将被认为假，语句块将被执行，然后再次对这个变量进行测试，整个过程就这样循环下去。

当然，写出一个无限循环的可能性是存在的。如果你写的 LISP 函数里有一个 **while** 循环，而你的终端在运行这个函数的时候又被挂起来了，那就极有可能是错误地写出了一个无限循环——这种错误并不少见。按下 “C-g” 组合键可以结束无限循环。

在我们的示例函数里，用来判断循环条件的函数是 “<”，这是一个需要有两个参数



的“小于”函数，它与 C 语言或 Pascal 语言中的“<”操作符的作用是一样的。第一个参数又是一个函数，它负责返回插入点的当前字符位置；第二个参数返回的是编辑缓冲区里的最大字符位置，也就是编辑缓冲区的长度。“<”函数（以及其他条件比较函数）返回的都是一个布尔值，即“t”或“nil”。

while 循环的语句块由两条语句组成。第 5 行把插入点向前移动一个单词（相当于按下“**ESC f**”组合键）。第 6 行给循环计数器加上一个 1，函数“**1+**”是“**(+ 1 variable-name)**”的简写形式。注意第 6 行上的第 3 个右括号是与单词 **while** 前面的那个左括号配对的。这个 **while** 循环会使 Emacs 以每次一个单词的间隔走过当前编辑缓冲区，同时完成统计单词个数的工作。

示例函数中的最后一条语句利用内部函数 **message** 在辅助输入区里显示一条消息，把编辑缓冲区里有多少个单词报告出来。C 程序员对 **message** 函数的格式应该是比较熟悉的。**message** 函数的第一个参数是一个输出格式字符串，它是由普通文本和“%x”形式的排版指令组成的，其中“x”是下列几个可用的字母之一。**message** 函数将依次把格式字符串中的排版指令套用到后续的参数上，并根据百分号后面的字母对相应的后续参数做出解释。下面是被用做排版指令的各个字母的含义：

%s	字符串或符号
%c	字符
%d	整数
%e	用科学计数法表示的浮点数
%f	用十进制小数表示的浮点数
%g	用最短字符串表示的浮点数

比如，下面这条语句：

```
(message "%s\\ is a string, %d is a number, and %c is a character"
         "hi there" 142 ?q)
```

将把下面这条消息：

```
"hi there" is a string, 142 is a number, and q is a character
```

显示在辅助输入区里。它相当于下面这条 C 语言代码：

```
printf ("\\"s\" is a string, %d is a number, and %c is a character\n",
       "hi there", 142, 'q')
```

浮点格式的输出字符情况稍微有些复杂。如果没有特别指定，它们将保留到小数点后面的6位数字。比如下面这条语句：

```
(message "This book was printed in %f, also known as %e." 1994 1994)
```

将产生如下所示的输出：

```
This book was printed in 1994.000000, also known as 1.994000e+03.
```

可以对小数点后面的保留位数进行设定，具体做法是在百分号“%”和字母“e”、“f”或“g”之间插入一个小数点和打算保留的小数位数。比如下面这条语句：

```
(message "This book was printed in %.3e, also known as %.0f." 1994 1994)
```

将在辅助输入区里显示下面这条消息：

```
This book was printed in 1.994e+03, also known as 1994.
```

把 LISP 函数转变为 Emacs 命令

我们刚才编写的 **count-words-buffer** 函数确实可以工作，但它还有几个小问题。如果已经把它输入进去，就可以在 LISP 交互窗口里输入 “(**count-words-buffer**)” 以执行之。把光标移动到该函数最后一个右括号的后面再按下“C-j”（或 **LINEFEED**）组合键，即 LISP 交互模式下的“求值”键就可以运行它了。

如果窗口里的光标（即按下“C-j”组合键的位置）后面还有一些文本，就会看到这个函数会让 Emacs 把插入点移动到编辑缓冲区的最末尾，它不会再返回到它原来的位置。这是 Emacs LSIP 命令普遍存在的一个问题，这个问题有个简单的解决办法，那就是 **save-excursion** 函数。这个函数的语法定义如下所示：

```
(save-excursion
  statement-block)
```

用了这个函数之后，语句块中的语句在运行时引起的光标移动在 Emacs 的内部进行，不会显示在屏幕上；当语句块全部执行完毕之后，插入点和文本块标记将出现

在它们原先的位置。除了预防出现刚才讲的不便以外，在LISP函数有多个嵌套层次的情况下，**save-excursion** 函数还可以作为一个“堆栈”来保存、和检索各嵌套层次的插入点和文本块标记的位置。**save-excursion** 函数的返回值是语句块中最后一条语句的返回值。

因此，如果你不想让用户看到某个函数中的光标移动命令把光标在屏幕上移来移去的，就可以把它们放在一个**save-excursion** 块里，比如下面这样：

```
(defun count-words-buffer ()
  (save-excursion
    (let ((count 0))
      (goto-char (point-min))
      (while (< (point) (point-max))
        (forward-word 1)
        (setq count (1+ count)))
      (message "buffer contains %d words." count))))
```

现在，既然已经能够让这个函数在LISP交互窗口里正确地执行，不妨像执行其他Emacs命令那样用“**ESC x**”执行这个函数。输入“**ESC x count-words-buffer RETURN**”命令，可看到的却是一条出错消息 “[No match] (不匹配)”。出现这条出错消息的原因是还没有把这个函数“登记”到Emacs里（必须先对函数进行注册才能在LISP交互模式里使用）。用来完成这项工作的函数是**interactive**，下面是它的语法定义：

```
(interactive "prompt-string")
```

这条语句必须是函数里的第一条语句，也就是说，它必须紧跟在包含**defun**和文档字符串（参见下一页）的那个代码行的后面。**interactive** 的作用是把函数注册为 Emacs 的一个命令，这样，当执行它的时候，Emacs 会提示用户输入你在**defun**语句里声明的参数。这个函数中的提示字符串是可选的。

提示字符串的格式有特殊要求：你必须为每一个你想提示用户输入的参数准备一个提示字符串的子字符串。子字符串之间要用**LINEFEED** (\n) 字符来分隔。每个子字符串的第一个字母必须是一个用来表示该参数类型的代码。参数类型代码有很多选择，我们把比较常用的列在表 13-1 里。



表 13-1：交互式函数的参数类型代码

参数类型代码	参数类型
b	一个现有编辑缓冲区的名字
e	事件（鼠标动作或功能键动作）
f	一个现有文件的名字
n	数字（整数）
s	字符串

这些代码还有相应的大写变体：

B	一个可能不存在的编辑缓冲区的名字
F	一个可能不存在的文件的名字
N	如果命令在调用时带有一个前缀参数，按前缀参数指示的情况办理；否则就是数字
S	符号

对“b”和“f”选项来说，当用户给出的编辑缓冲区或文件不存在时，Emacs会报告出错。**interactive**函数还有一个很有用的选项是“r”，我们稍后再对它进行介绍。我们还将在第十四章介绍一个“e”选项。还有很多其他的选项字母，具体细节请查阅**interactive**函数的有关文档。子字符串的剩余部分是显示在辅助输入区里的实际内容。

用**interactive**来填充函数参数的具体方法是比较复杂的，我们最好还是用一个例子来做一下说明。函数**goto-percent**（我们马上就会看到它）里有一个简单的例子，它包含的语句是：

```
(interactive 'nPercent: "")
```

提示字符串里的字母“n”告诉Emacs提示用户输入一个整数；字符串“Percent:”是实际出现在辅助输入区里的提示文字。

再来看一个稍微复杂点的例子。假设我们想写出一个自己的**replace-string**命令。下面是我们写的用户提示部分：

```
(defun replace-string (from to)
  (interactive 'sReplace string: `\\nsReplace string %s with: ')
  ...)
```

这个提示字符串是由两个子字符串组成的，一个是“`sReplace string:`”，另一个是“`sReplace string %s with:`”，两个子字符串之间用一个 **LINEFEED** 字符隔开。两个子字符串的第一个字母都是“`s`”，这表示预期用户输入字符串：“`%s`”是一个格式操作符（参见前面介绍 **message** 函数的有关内容），Emacs 会用用户在第一条提示下输入的响应替换它。

当这个命令被调用的时候，首先出现在辅助输入区里的是第一条提示信息“`Replace string:(置换字符串)`”。假设用户输入“`fred`”作为响应。当用户按下回车键之后，第二条提示信息“`Replace fred with:`”将出现在辅助输入区里。用户接着输入替换字符串并再次按下回车键。

用户输入的两个字符串将（按其输入的先后顺序）被分别用做函数参数“`from`”和“`to`”的值，整个命令将继续执行直到结束。这就是说，**interactive** 是按照提示字符串中子字符串的先后顺序来为函数参数提供值的。

interactive 不仅能够作为独立的函数来使用，我们还可以从其他 LISP 代码里对这个函数进行调用；在这种场合里，对它进行调用的那个函数必须负责为它准备好全体参数的值。请看下面的例子。如果我们想在另外一个 LISP 函数里调用刚才编写的 `replace-string` 命令，以便把某个文件里出现“`Bill`”的地方都替换为“`Deb`”，就可以用下面这条语句来实现：

```
(replace-string "Bill" "Deb")
```

在这个例子里，因为函数并不是在交互方式下被调用的，所以 **interactive** 语句没有任何作用；参数“`from`”被设置为字符串“`Bill`”，参数“`to`”被设置为字符串“`Deb`”。

再回到我们的 **count-words-buffer** 命令上来：它本身没有参数，所以它里面的 **interactive** 命令也就不需要有提示字符串。最后，我们还想给此命令增加一个“文档字符串”（*documentation string*，或者简写为“*doc string*”），这是用户发出 **describe-function**（“C-h f”组合键）等在线求助命令时将会看到的帮助信息。文档字符串是正常的 LISP 字符串；它们是可选的，其长度可以是任意多行。但传统上，它的第一行应该是用来对命令功能进行介绍的一个完整而又精练的句子。需要提醒大家注意的是：字符串里的双引号（“）前面必须加上一个反斜线字符（\）作为前缀。



在综合考虑了以上介绍的各种因素之后，我们最终得到的函数是下面这样的：

```
(defun count-words-buffer ()
  "Count the number of words in the current buffer;
print a message in the minibuffer with the result."
  (interactive)
  (save-excursion
    (let ((count 0))
      (goto-char (point-min))
      (while (< (point) (point-max))
        (forward-word 1)
        (setq count (+ 1 count)))
      (message "buffer contains %d words." count))))
```

LISP 语言的基础函数

在介绍完自行编写工作函数的方法之后，我们再来看看LISP语言的基础函数。它们是你用来打造自己的工作函数的建筑材料。正如我们前面讲的那样，LISP在其他程序设计语言使用操作符（用于算术运算、比较运算和逻辑运算）的地方使用的是函数。表 13-2 列出了一些相当于其他程序设计语言中各种操作符的 LISP 基础函数。

表 13-2：LISP 语言的基础函数

算术运算	+、-、*、/（加、减、乘、除） %（求余数） 1+（递增） 1-（递减） max（最大值）、min（最小值）
比较运算	>、<、>=、<=（大于、小于、不小于、不大于） /≠（不等于） =（等于，用于数字和字符） equal（等于，用于字符串和其他复杂的数据对象）
逻辑运算	and（与）、or（或）、not（非）

除“1+”、“1-”和“%”以外的其他算术函数都可以有任意多个参数；逻辑函数“and”和“or”也是如此。算术函数只有在其参数中至少有一个是浮点数的时候才会返回浮点值；也就是说，函数(/ 7.0 2)的返回值是 3.5，而(/ 7 2)的返回值是 3。注意整数除法将舍去余数。



有些工作用函数来做未免效率不高，而且满纸全都是函数的程序也很难看。可话又说回来，LISP 语言的主要优点之一就是它的核心代码少而精，高效率的解释执行很容易实现。而且，有 Emacs 的各种 LISP 模式作为程序设计支持工具，语法也不会是一个多么严重的问题。

语句块

我们见过用 **let** 函数定义的语句块，也见过 **while** 结构里的语句块。LISP 语言中还有其他一些能够定义语句块的构造：**progn** 和 **let** 的其他形式。

progn 是最基础的，它的语法定义是：

```
(progn  
  statement-block)
```

progn 是个让一个语句块看起来就像是一条语句的简单方法，它的作用与 Pascal 语言中的 **begin** 和 **end** 或者 C 语言中的花括号有些相似。**progn** 的返回值就是语句块中最后一条语句的返回值。**progn** 特别适合用在像 **if** 这样的控制结构里（参见下面的内容），因为这些控制结构不像 **while** 那样允许使用语句块。

let 函数还有其他一些变体形式。其中最简单的是：

```
(let (var1 var2 ...)  
  statement-block)
```

上面这个语法定义没有使用 “**(var value)**” 这样的列表项，它只有一个由变量名组成的列表。和 **let** 函数的其他形式一样，用这种方法定义的变量也是一些局部变量，只能在这个语句块里访问。因为没有把它们初始化为给定的值，所以这些变量都被初始化为 “**nil**”。在实际工作中，完全可以把这两种形式的 **let** 函数混在一起使用，比如像下面这样：

```
(let (var1 (var2 value2) var3 ...)  
  statement-block)
```

在我们见到的第一种 **let** 格式里，局部变量的初始值可以是函数调用（记住，函数都有返回值）。在开始对任何一个变量进行赋值之前，必须先把这些函数的返回值全都求出来。但有些场合需要用某些局部变量的值，来计算另外一些局部变量的值，这就需要用到 **let** 函数的最后一一种形式——**let***。

我们来看一个例子。假设我们想编写一个 **goto-percent** 函数，它的作用是能够前进到用百分比表示的文本在编辑缓冲区里的某个位置上。下面是这个函数的一种写法：

```
(defun goto-percent (pct)
  (interactive "nGoto percent: ")
  (let* ((size (point-max))
         (charpos (/ (* size pct) 100)))
    (goto-char charpos)))
```

我们在前面已经讲过，**interactive** 函数是用来提示用户输入参数值的；在上面这个例子里，它将提示用户为参数 **pct** 输入一个整数值。接下来，**let*** 函数先把 **size** 初始化为编辑缓冲区里的字符总数，然后计算出一个字符位置 **charpos**，让 **charpos**、**size** 和 **pct** 正好是一个百分比的关系。最后，我们用 **goto-char** 函数把光标移动到当前窗口里的指定字符位置。

这段代码最重要的一点是：如果我们用 **let** 来代替 **let***，在需要计算 **charpos** 的值时，**size** 的值还没准备好。**let*** 也可以用在 **(var1 var2 ...)** 结构里，就像 **let** 一样。

下面是 **goto-percent** 函数更有效率的一种写法：

```
(defun goto-percent (pct)
  (interactive "nPercent: ")
  (goto-char (/ (* pct (point-max)) 100)))
```

控制结构

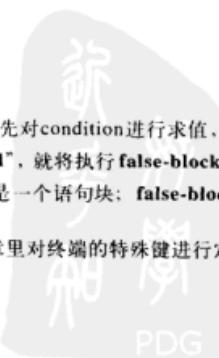
我们已经见过 **while** 函数用做控制结构的情况——与其他程序设计语言里类似语句的情况差不多。LISP 语言还有另外两种重要的控制结构，它们是 **if** 和 **cond**。

if 函数的语法定义是：

```
(if condition
    true-case
    false-block)
```

它的执行情况是这样的：先对 **condition** 进行求值，如果它不是 “nil”，就将执行 **true-case** 部分；如果它是 “nil”，就将执行 **false-block** 部分。**true-case** 必须是单独一条语句，而 **false-block** 则是一个语句块；**false-block** 可以省略。

我们来回忆一下第十一章里对终端的特殊键进行定制的例子。如果你使用着一个以



上的终端，就应该给 “*.emacs*” 文件增加一些代码，让它能够判断出你当前正使用着哪一个终端并以此为根据来设置相应的按键绑定。

这段代码同时还演示了其他几个重要的 Emacs LISP 功能的用法。为简单起见，我们的函数将以 DEC VT100 / 200 系列的 ANSI 终端和 X 窗口系统中的 **xterm** 终端仿真器等 3 大类终端上的方向键为目标进行设计。表 13-3 列出了各种方向键所输出的字符代码。

表 13-3: ANSI 方向键输出的字符代码

箭头键	字符代码
上	ESC O A
下	ESC O B
右	ESC O C
左	ESC O D

下面是用来设置键位映射图和按键绑定的有关代码。代码中的分号是注释标记；分号之后直到行尾的一切文本都是注释内容。如下所示：

```
(setq termtype (getenv "TERM"))

(if (or (equal termtype "vt200")           ; 这 3 种终端都有 ANSI 小键盘
        (equal termtype "vt100")
        (equal termtype "xterm"))

  (progn
    (setq term-file-prefix nil)           ; 覆盖默认的键位映射
    (send-string-to-terminal '\e=')       ; 强制应用小键盘
    (setq SS3-map (make-keymap))         ; 建立小键盘的键位映射
    (define-key global-map "\eO" SS3-map)

    (define-key SS3-map "A" 'previous-line) ; 上箭
    (define-key SS3-map "B" 'next-line)   ; 下箭
    (define-key SS3-map "C" 'forward-char) ; 右箭
    (define-key SS3-map "D" 'backward-char) ; 左箭

  (if (equal termtype other-term-type...)
    (progn
      code for binding keys on other terminal))))
```

第一条语句使用了内部函数 **getenv**，它是一个面向 UNIX 操作系统的接口，作用是把某给定环境变量的值（参见第十一章）以字符串的形式返回来。上面例子里使用

的 UNIX 环境变量 **TERM** 描述的是正在使用的终端的类型（如果你的终端设置正确……）。

接下来，**if** 子句的条件判断部分检查 **termtype** 是否是 3 个给定的终端类型之一。如果是，就将执行 **progn** 语句块里的代码。记住：**if** 函数的“真”部分只能有一条语句，所以我们在这里用 **progn** 定义了一个语句块。这个语句块中的前 4 条语句是我们在第十一章里见过的；而语句块里的其他语句则把方向键绑定到相应的光标移动命令上。

第二个 **if** 语句实际上是第一个 **if** 语句的“false（假）”子句。就如同 Pascal 或 C 语言中的 **else if**。在这个 **if** 子句中可以检测其他终端类型并在 **progn** 语句块中为该类型绑定按键。

更通用的条件控制结构是 **cond** 函数，它的语法格式如下所示：

```
(cond
  (condition1
    statement-block1)
  (condition2
    statement-block2)
  ...)
```

C 和 Pascal 程序员可以把 **cond** 函数看做是由一连串 “*if then else if then else if...*” 组成的语法结构；也可以把它看做是一组 “*case*” 或 “*switch*” 语句。**cond** 函数中的条件将按其出现的先后顺序依次被求值，如果某个条件的求值结果不是 “**nil**”，就将执行相应的语句块；而 **cond** 函数也将在执行这个语句块之后结束运行，它的返回值就是该语句块中最后一个语句的值（注 4）。

我们可以用 **cond** 函数重写前面的 **if** 函数示例。假设我们还是以刚才那 3 类不同的终端类型为目标，最后一个条件保留给缺省终端使用：

```
(setq termtype (getenv "TERM"))
(cond ((or (equal termtype "vt200")
           (equal termtype "vt100"))
        ;这 3 种终端都有 ANSI 小键盘
```

注 4： **cond** 函数里的语句块其实是可以省略的；有些程序员喜欢省略最后一个语句块，把最后一个“条件”留做它前面的其他条件计算值都是 “**nil**” 时的“例外”子句。如果省略语句块，则 **cond** 函数的返回值就简单地取为与之对应的条件的计算值。

```
(equal termtype "xterm"))
(setq term-file-prefix nil)           ; 覆盖默认的键位映射
(send-string-to-terminal "\e=")        ; 强制应用小键盘
(setq SS3-map (make-keymap))          ; 建立小键盘的键位映射
(define-key global-map "\eO" SS3-map)

(define-key SS3-map "A" 'previous-line) ; 上箭
(define-key SS3-map "B" 'next-line)    ; 下箭
(define-key SS3-map "C" 'forward-char) ; 右箭
(define-key SS3-map "D" 'backward-char) ; 左箭

((equal termtype other-term-type ...))
  (code for binding keys on other terminal))

(t
  (code for binding keys of third terminal type)))
```

第3个条件表达式很简单，就是一个原子项“t”（真）；它的含义是：如果终端不属于前两大类，就认定它是第3种（缺省的）终端类型并执行相应的代码。

Emacs 的内部函数

许多现成的Emacs函数和一些自行开发的函数都涉及到对编辑缓冲区里的文本进行查找和替换之类的操作。这些函数在某些特定的编辑模式（比如RMAIL模式和上一章介绍的各种语言编辑模式）里非常有用。Emacs有许多内部函数都与字符串和编辑缓冲区里的文本的处理工作有关；其中最值得一提的是那些利用Emacs的正则表达式功能的函数。对正则表达式的介绍请参考第三章。

我们先介绍一些与编辑缓冲区和字符串有关却又没有使用正则表达式的Emacs基本函数。然后，我们将在第三章内容的基础上对正则表达式做更深入的研究；讨论重点将集中在对LISP程序设计最有帮助的那些功能上。最后，我们将介绍一些对正则表达式本身进行操作的Emacs函数。

编辑缓冲区、文本和文本块

表13-4列出了一些与编辑缓冲区、文本和字符串有关的Emacs基本函数，它们是一些通常只有LISP程序员才会用到的函数，所以没有被绑定到任何按键上。我们已经在刚才编写的**count-words-buffer**函数里见过它们当中的几个了。

表 13-4：与编辑缓冲区和文本有关的函数

函数名称	返回值或执行动作
point	光标的字符位置
mark	文本块标记的字符位置
point-min	最小字符位置（通常是 1）
point-max	最大字符位置（通常是编辑缓冲区的长度）
bolp	光标是否位于行首（取值为“t”或“nil”）
eolp	光标是否位于行尾
bobp	光标是否位于编辑缓冲区的开始
eobp	光标是否位于编辑缓冲区的末尾
insert	把任意个数的参数（字符串或字符）插入到编辑缓冲区光标位置之后
number-to-string	把一个数值参数转换为一个字符串
string-to-number	把一个字符串转换为一个数字（整数或浮点数）
char-to-string	把一个字符参数转换为一个字符串
concat	把任意个数的字符串合并到一起
substring	给定一个字符串及两个整数索引 <i>start</i> 和 <i>end</i> ，返回从 <i>start</i> 指示的位置开始到 <i>end</i> 指示的位置前结束的子字符串。下标从 0 开始计算。比如说，函数“(substring “appropriate” 2 5)”将返回子字符串“pro”
aref	数组索引函数，它可以用来从字符串里取出单个的字符；它的输入参数是一个整数，从函数返回的是一个以整数表示的字符。返回值（在大多数机器上）使用的是 ASCII 码。比如说，函数“(aref “appropriate” 3)”将返回数字“114”，即字母“r”的 ASCII 码

很多没有出现在上面这个表格里的函数——包括一些大家都很熟悉的用户级命令在内——也能对编辑缓冲区和文本进行处理。有几个比较常用的 Emacs 函数是以文本块 (*region*, 即编辑缓冲区中的文本区域) 为操作对象的。在使用 Emacs 做文字编辑工作时，要用先设置文本块标记、再移动光标的方法来选取文本块。可如果想用面向文本块的函数（比如 **kill-region**、**indent-region** 和 **shell-command-on-region** 等等）。总之，任何名字里带有“*region*”字样的函数）来编写 Emacs LISP 代码，就应该采用一种更灵活的方法来选取文本块——这些函数也确实有这种灵活性。面向文本块的函数一般都有两个整数参数，它们的作用是给出文本块前后边界的字符位置；如果函数是在交互方式下调用的，这两个参数的缺省值将是 **point** 和 **mark** 函数的返回值。

很明显，让 Emacs 自动把 point 和 mark 函数的返回值设置为这些参数的缺省值，要比由用户自己动手来输入这些值方便得多（也更符合人们的习惯）。**interactive** 函数的“r”选项使这种设想成为现实。我们来看一个例子。我们打算编写一个能够把文本块翻译为德文的**translate-region-into-German** 函数，于是，写出下面这些代码：

```
(defun translate-region-into-German (start end)
  (interactive "r")
  ...)
```

当这个函数以交互方式调用时，**interactive** 的“r”选项将自动把参数 **start** 和 **end** 的值设置好；但如果从其他 LISP 代码里对它进行调用，就必须给出这两个参数。在 LISP 代码里调用这个函数最常见的方法是：

```
(translate-region-into-German (point) (mark))
```

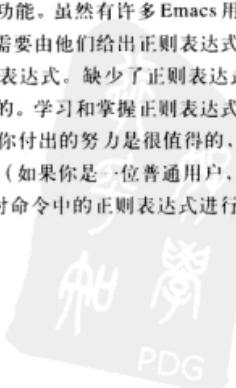
事实上，并不是非得像这样来调用它。如果想用这个函数写一个用来把整个编辑缓冲区翻译为德文的**translate-buffer-into-German** 函数，就只需给它加上一个下面这样的“包装”：

```
(defun translate-buffer-into-German ()
  (translate-region-into-German (point-min) (point-max)))
```

事实上，除非是确有必要，否则最好不要在 LISP 代码里直接使用 point 和 mark 函数；应该使用局部变量来选取文本块。写出的 LISP 函数不应该是 Emacs 编辑命令的堆砌——后者更适合用 Emacs 的宏（参见第十章）来实现。

正则表达式

正则表达式（简称 regexp）提供了更强大的文本处理功能。虽然有许多 Emacs 用户不喜欢使用 **replace-regexp** 和 **re-search-forward** 等需要由他们给出正则表达式的命令，但编写 LISP 代码时却经常要大量地使用正则表达式。缺少了正则表达式，RMAIL、Dired 以及各种语言编辑模式将是无法想像的。学习和掌握正则表达式的使用方法需要时间和耐心，但作为一个 LISP 程序员，你付出的努力是很值得的，因为你程序里的正则表达式只能由你本人来编写和调试（如果你是一位普通用户，使用的是用户级的 Emacs 编辑命令，就不需要也不能对命令中的正则表达式进行调试）。



我们将用一些查找和替换方面的例子来介绍正则表达式的各种功能；这些例子都是比较容易理解的，没有过多地纠缠在晦涩难懂的细节方面。在此之后，我们将介绍几个高级的LISP函数，它们能够用正则表达式完成更加复杂的查找和替换操作。在下面这几个需要使用查找和替换操作的情况里，简单的查找/替换命令或者根本无法完成，或者完成得不好：

1. 在用C语言开发代码，想把例程 `read` 和 `readfile` 的功能合并到一个名为 `get` 的新函数里去。需要把程序代码里凡是出现“`read`”和“`readfile`”的地方都替换成“`get`”。
2. 按第八章介绍的方法用Emacs的大纲模式撰写着一篇 `troff` 文档。在大纲模式下，文档中的段落标题是一些以一个或者多个星号（*）开始的文本行；可是并不想让这些星号也出现在文档的打印稿里。于是，决定自己写一个 `remove-outline-marks` 函数去掉文档里的这些星号，然后再用 `troff` 对文件进行排版。
3. 想把文档里的“`program`”、“`programs`”和“`program's`”分别替换为“`module`”、“`modules`”和“`module's`”，但不能让“`programming`”和“`programmer`”也类似地被替换为“`moduleming`”和“`modulemer`”。
4. 正在为以Ada语言重写的某个C语言软件编制文档。因为Ada编译器要求使用“.a”做为文件名后缀，所以需要把文档里的文件名都从“`filename.c`”改为“`filename.a`”。
5. 新安装的C++编译器产生的出错信息是德文。于是，想修改Emacs的 `compile` 程序包，好让它能够正确地对出错信息进行分析（参见第十二章结尾部分）。

我们马上就会介绍利用正则表达式来解决上面这些问题的方法，为了便于查阅，我们给这些例子都编了号。请大家注意，虽然这一章对正则表达式的讨论比第三章的内容更加深入，但并没有涉及到它的全部功能；那些没有在此介绍的功能或者与我们介绍的功能有所重叠，或者与本书的讨论范围没有直接的关系。更需要大家注意的是，这里探讨的正则表达式语法只适用于LISP字符串；我们很快就会看到，LISP字符串所使用的正则表达式语法与用户级 Emacs 命令（比如 `replace-regexp`）所使用的正则表达式语法两者之间有一个很重要的区别。

基本操作符

正则表达式最初只是计算机科学在理论方面的一个想法，但它如今已经深入到计算

领域的每一个角落。用来表示它们的语法可能会千差万别，但其核心都是一样的。某些正则表达式的表示法应该是大家都很熟悉的了，比如UNIX操作系统的shell用来匹配文件名的那些通配符。但Emacs采用的表示方法与UNIX的文件名通配符稍微有些区别；它与ed和vi等编辑器以及lex和grep等UNIX软件工具所使用的记号方法比较接近。我们先从与UNIX的shell通配符功能相近的Emacs正则表达式操作符开始讲起，这些操作符都列在表13-5中。

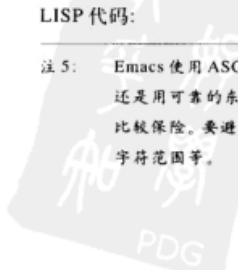
表 13-5：正则表达式基本操作符

Emacs 操作符	UNIX shell 操作符	作用
.	?	匹配任意一个字符
.*	*	匹配任意一个字符串
[abc]	[abc]	匹配字母 a、b 或 c
[a-z]	[a-z]	匹配任意一个小写字母

比如，如果想匹配所有以“program”开头的文件名，在UNIX shell里得使用“program*”；在Emacs里就得使用“program.*”。如果想匹配所有以字母“a”到“e”开头的文件名，在UNIX里要使用 “[a-e]*”或 “[abcde]*”；在Emacs里就要用 “[a-e].*”或 “[abcde].*”。换句话说，方括号里的短划线字符（-）代表字符范围（注5）。我们过一会儿再来对字符范围和允许放在方括号里的字符集做进一步的探讨。

如果需要在正则表达式里放上一个本身被用做操作符的字符，就必须在它的前面加上两个连续的反斜线字符，比如“*”将匹配一个星号字符。为什么要使用两个反斜线字符呢？答案是这与Emacs LISP读取和解码字符串的操作方法有关。在把一个字符串读到一个LISP程序里的时候，Emacs会对用反斜线字符转义的字符进行解码，即把两个反斜线字符转换为一个单个的反斜线字符。如果字符串被用做一个正则表达式——即当它被传递到某个需要正则表达式作为参数的函数里的时候，那个函数会使用这个单个的反斜线字符作为正则表达式语法的一部分。请看下面这行LISP代码：

注 5： Emacs 使用 ASCII 码（在大多数机器上）来确定字符范围，但你可别依赖这个事实；还是用可靠的东西——比如小写字符集、大写字符集或用来表示数字的 “[0-9]” 比较保险。要避免使用不具备移植性的表示方法，比如 “[A-Z]” 和包含标点符号的字符范围等。



```
(replace-regexp "fred\\*\" \"bob\")
```

LISP解释器会先把字符串“fred*”解码为“fred*”，然后再把它传递到**replace-regexp**命令。**replace-regexp**命令知道“fred*”表示的是“fred”后面加一个星号。注意，**replace-regexp**命令的第二个参数不是一个正则表达式，所以“bob*”中的星号不需要用反斜线字符进行转义。还要注意的是：如果把这条语句当做一个用户级命令来调用，就不需要输入两个反斜线字符。也就是说，只需输入“**ESC x replace-regexp RETURN**”，然后是“**fred***”和“**bob***”。Emacs采用另外一种方法来解码从辅助输入区里读入的字符串。

Emacs里的“*”正则表达式操作符与UNIX shell使用的“*”通配符含义是不一样的：它的含义是“星号（*）前面任何东西的零次或多次出现”。于是，因为“.”能匹配任意一个字符，“.*”就表示“任意字符的零次或多次出现”，即任意一个字符串。“*”前面可以是任何东西：比如，“read*”将匹配“rea”后面跟有零个或者多个字母“d”的情况；“file[0-9]*”将匹配“file”后面跟有零个或者多个数字的情况。

有两个操作符与“*”的关系非常密切。第一个是“+”，它将匹配“+”前面任何东西的一次或者多次出现。于是，“read+”将匹配“read”和“readdddd”，但不匹配“rea”；而“file[0-9]+”要求“file”后面至少要跟有一个数字。第二个是“？”，它的含义是“？”前面任何东西的零次或者一次出现（即使之成为可选项）。于是，“read?”将匹配“rea”和“read”；而“file[0-9]?”则要求“file”后面最多跟有一个数字。

在介绍其他操作符之前，我们先说说字符集和字符范围。首先，允许在一个字符集中指定一个以上的字符范围。因此，字符集“[A-Za-z]”代表的是字母表中的全体字符；这比不具备移植性的“[A-z]”要好得多。把字符范围和字符集中的字符的列表组合在一起也是允许的；比如，字符集“[A-Za-z]_”代表的是字母表中的全体字符加下划线字符，这正是允许用在C语言标识符名称里的全体字符。如果在某字符集第一个字符的前面加上字符“^”，它的作用就相当于“not（取反）”操作符；该字符集将匹配所有没有出现在“^”字符之后的那些字符。比如，字符集“^[A-Za-z]”将匹配全体非字母表字符。

如果字符“^”没有出现在字符集第一个字符的前面，它就没有特殊的含义；它只是

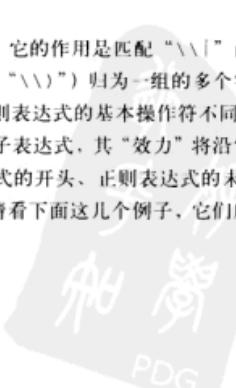
一个上箭头罢了。与此相对的是，如果短划线字符（-）出现在字符集的最开始，它就将失去特殊含义；右方括号字符（]）也遵守同样的规定。但我们不推荐把这些用做操作符的字符放在字符集里——这是一种投机取巧的做法；为安全起见，最好是把它们放在字符串里，而且要用两个反斜线字符对它们进行转义。给一个非特殊字符加上一个双反斜线前缀得到的通常还是那个字符——但最好别这么做！正则表达式操作符还包括其他一些字母和标点符号，我们将在以后的内容里再介绍几个。有些字符在用双反斜线前缀进行转义后会变成操作符，我们也会在后面的内容里把它们介绍给大家。“^”字符如果被放在字符集的外面会有另外一种含义，我们马上就会看到。

归组操作符和二选一操作符

如果想让“*”、“+”或“?”对一个以上的字符进行操作，就需要使用“\\(|”和“\\)|”操作符对字符串进行归组。注意，在这两个操作符（以及此后介绍的其他操作符）里，那两个反斜线字符是操作符的组成部分。（为了避免出现过多的“特殊”字符，除了“*”、“+”或“?”等正则表达式基本操作符之外，其他一切字符（包括反斜线字符在内）都必须遵守这条规定。）大家刚才也看到了，如果想让“*”、“+”或“?”作为字符出现在字符串里，就必须用两个反斜线字符对它们进行转义，这样才能使 Emacs 能够对它们正确地解码。如果某个基本操作符紧跟着出现在“\\)|”的后面，它就将把“\\(|”和“\\)|”之间的内容作为一个整体来对待。比如，“\\(read\\)*”将匹配空字符串、“read”和“readread”等等；而“read\\(file\\)?”将匹配“read”或“readfile”。现在，我们可以开始着手解决这一节开始时列出来的“疑难杂症”了。下面这行 LISP 代码将解决“情况 1”：

```
(replace-regexp "read\\(file\\)*" "get")
```

二选一操作符“\\|”是一个“非此即彼”操作符；它的作用是匹配“\\|”前后两样事物中的一个。对于用归组操作符（“\\(|”和“\\)|”）归为一组的多个字符（即字符组）来说，“\\|”操作符对它们的处理与正则表达式的基本操作符不同。它并不挑剔字符组是否是由一个以上的字符所组成的子表达式，其“效力”将沿它自己的左、右方向伸展到极至，直到它到达正则表达式的开头、正则表达式的末尾、一个“\\(|”、一个“\\)|”或者另外一个“\\|”。请看下面这几个例子，它们应该能把问题说得更清楚：



- “read\\|get”将匹配“read”或“get”
- “readfile\\|read\\|get”将匹配“readfile”、“read”或“get”
- “\\(read\\|get\\)file”将匹配“readfile”或“getFile”

在第一个例子里，“\\|”的效力伸展到了正则表达式的开头和末尾。在第二个例子里，第一个“\\|”的效力向左伸展到了正则表达式的开头，向右伸展到了第二个“\\|”。在第三个例子里，“\\|”的效力向左、向右分别伸展到了“\\(”和“\\)”。

与上下文有关的正则表达式操作符

正则表达式另一组重要的操作符与字符串的上下文有关，所谓“上下文 (context)”指的是字符串前后的文本。我们在第三章讨论递增查找操作时介绍过一个 **word-search** 命令，它在执行时就需要依赖于特殊的上下文设置；**word-search** 命令所依赖的上下文就是出现在字符串前、后的单词分隔字符，比如空格和标点符号等。

正则表达式中最简单的上下文操作符是“^”和“\$”，它们也是正则表达式的基本操作符，使用时必须分别放置在正则表达式的开头或者末尾。“^”操作符的作用是让正则表达式的后续部分只匹配它出现在行首的情况；而“\$”操作符则是让正则表达式的前导部分只匹配它出现在行尾的情况。在“情况 2”里，我们需要让函数匹配一个或者多个星号出现在行首的情况，下面这段代码将解决这一问题：

```
(defun remove-outline-marks ()
  "Remove section header marks created in outline-mode."
  (interactive)
  (replace-regexp "^ \\*+ \"") )
```

这个函数会找出那些以一个或者多个星号开头的文本行（“*”表示匹配“*”字符本身，“+”表示“一个或一个以上”），然后它会把那些星号替换为空字符串（“”）——也就是删除它们。

请注意，如果想匹配的字符串可能延伸到两个或者更多个文本行，千万不要把“^”和“\$”放在正则表达式的半中间，这是不允许的。正确的做法是在正则表达式里的适当位置加入“\\n”（作用是匹配 **LINEFEED** 字符）来匹配这样的字符串。另一个有类似功效的字符是用来匹配制表符 **TAB** 的“\\t”。当“^”和“\$”出现在对字符串（而不是对编辑缓冲区中的文本行）进行查找的正则表达式里时，它们将分别匹

配字符串的开头和结尾；本章后面将要介绍的**string-match**函数能够对字符串进行正则表达式查找。

下面这个复杂的正则表达式是一个实实在在的例子，它里面用到了我们此前介绍的各种操作符：**sentence-end**——Emacs 用来识别句子结尾的变量，诸如**forward-sentence**（“**ESC e**”组合键）等，以句子为单位的光标移动命令就是靠这个变量来完成操作的。它的值是：

```
"[.?!][\\n^"])*\\($\\|\\t\\n\\|\\r\\n\\)*"
```

让我们逐段分析一下这个正则表达式。第一个字符集即“[.?!]”，能匹配一个英文句号、一个问号或者一个感叹号（英文句号字符“.”和问号字符“?”也是正则表达式的操作符，但它们在字符集里没有特殊含义）。接下来是“[\\n^"])*”，这个字符集里包含有右方括号、双引号、单引号、右圆括号和右花括号。紧跟在这个字符集后面的星号（*）表示匹配这个字符集中任何一个字符的一次或者多次出现的情况。截止到这里，这个正则表达式将匹配一个表示句子结束的标点符号，以及一个或者多个出现在句尾标点符号后面的引号、括号或者花括号。再往后是一个用来进行多选一匹配的字符组“\\(\$\\|\\t\\n\\|\\r\\n\\)”，它将匹配三样东西：“\$”（行尾符）、制表符 **TAB** 或两个空格中的任何一个。最后，“[\\t\\n]*”匹配零个或者多个空格、制表符或换行符 **LINEFEED**。也就是说，句子结尾字符后面还可以有行尾符、或者任意组合的空格（至少两个）、制表符和换行符 **LINEFEED**。

上下文操作符并非只有“^”和“\$”两个；我们再介绍两个可以利用正则表达式来进行单词查找的上下文操作符。操作符“\\<”和“\\>”分别匹配单词的开始和结尾。有了这两个操作符，我们就可以把“情况3”里的问题解决一大半了。正则表达式“\\<program\\>”将匹配“program”，但不匹配“programmer”或“programming”（它也不会匹配上“microprogram”）。问题差不多解决了；但它还不能匹配上“program's”或“programs”。为了把这些情况都匹配上，我们需要一个更复杂的正则表达式：

```
\\<program\\>('s\\|s\\)?\\>
```

（出现在正则表达式最后面的“\\>”可以省略。）这个正则表达式的含义是“一个以 **program** 开始，后面可能还跟有一个's或s的单词”。这个正则表达式已经把“情况3”在匹配正确单词方面的问题都解决了。

截取匹配目标的部分内容

离成功只差一步了：“program”后面可能带有“s”或“'s”，可我们的正则表达式现在还无法做到在不触及“s”或“'s”的同时把“program”修改为“module”。这就引出了我们准备在这一章里探讨的最后一个正则表达式功能：截取匹配目标（即匹配操作找到的字符串）的部分内容以便对之做进一步处理。我们刚才得到的正则表达式本身是正确的，但它只是**replace-regexp**命令的查找字符串而已。至于替换字符串，答案是“module\\1”；换句话说，完整的LISP代码应该是：

```
(replace-regexp "\\\<program\\(\s\\|s\\\\)?\\>" "module\\1")
```

从最终效果上讲，“\\1”的含义是“只对匹配目标中与“\\(”和“\\)”之间的子表达式相匹配的部分内容进行替换”。这是替换操作中惟一允许使用的与正则表达式有关的操作符。就“情况3”而言，它的含义是：如果匹配目标是“program's”，就先把替换字符串中的“\\1”替换为“'s”，然后再对匹配目标进行替换；如果匹配目标是“programs”，就得先把替换字符串中的“\\1”替换为“s”；如果匹配目标只是“program”，就先把替换字符串中的“\\1”替换为空字符（即去掉替换字符串中的“\\1”）。最终的结果是皆大欢喜：“program”将被替换为“module”、“programs”将被替换为“modules”、“program's”将被替换为“module's”。

“情况4”也需要用这个功能来解决。为了匹配“filename.c”并把它替换为“filename.a”，我们需要使用下面的LISP代码：

```
(replace-regexp '\\(filename\\\\)\\.c" "\\1.a")
```

注意，“\\.”代表的是一个“真正”的英文句号(.)；而给查找字符串里的“filename”两端加上“\\(”和“\\)”的惟一目的是为了让替换字符串里的“\\1”能够截取出句号来。

事实上，“\\1”操作符只是正则表达式一种更强大的功能的一个特例。一般说来，如果用“\\(”和“\\)”对正则表达式里的某一小段进行了归组，那么，与这个子表达式相匹配的字符串就会被保存起来；接着，当设计替换字符串的时候，就可以用“\\N”来截取那些被保存起来的子字符串，其中“N”是被归组的子字符串的编号，字符组从左向右顺序编号，编号从1开始。被归组的表达式可以嵌套；与之对应的“\\N”编号将按“\\(”出现的先后顺序从左向右进行赋值。

完整地使用这项功能的LISP代码，通常都包含着相当复杂的正则表达式。Emacs自

身的LISP代码里就有一个绝佳的例子，它就是（我们在上一章快结束时介绍的）**compile**程序包在分析编译器给出的出错信息时，所使用的正则表达式列表**compilation-error-regexp-alist**。我们把这个列表变量从Emacs的源代码里节选出来列在下面，有兴趣的读者可以好好研究一下：

```

;; 4.3BSD grep, cc, lint pass 1:
;;   /usr/src/foo/foo.c(8): warning: w may be used before set
;; or GNU utilities:
;; foo.c:8: error message
;; or HP-UX 7.0 fc:
;; foo.f          :16      some horrible error message
(*\n\\ ([^:( \t\\n]+\\)[:]{}[\t]*\\([0-9]*\\)[:] \t!` 1 2)

;; 4.3BSD lint pass 2
;; strcmp: variable # of args. llib-1c(359)  ::  /usr/src/foo/foo.c(8)
(*\t:\\)\\([^(:( \t\\n]+\\)[:]{}(+[\t]*\\([0-9]*\\)[:] )*` 1 2)

;; 4.3BSD lint pass 3
;; bloofle defined( /users/wolfgang/foo.c(4) ), but never used
;; This used to be
(*{(\t)*\\([^(:( \t\\n)+\\)[:]{}[\t]*\\([0-9]*\\)`)` 1 2)

;; Ultrix 3.0 f77:
;; Error on line 3 of t.f: Execution error unclassifiable statement
;; Unknown who does this:
;; Line 45 of "foo.c": bloofel undefined
(*\n\\ (Error on \\)?Line[ \t]+\\([0-9]*\\)[ \t]+\\off[ \t]+\\?\\({^"\\"n]+\\)\\?` 3 2)

;; Apollo cc, 4.3BSD fc:
;; 'foo.f', line 3: Error: syntax error near end of statement
;; IBM RS6000:
;; vvouch.c', line 19.5: 1506-046 (S) Syntax error.
;; Unknown compiler:
;; File "foobar.ml", lines 5-8, characters 20-155: blah blah
(*"\\"\\([^\\"\\n]+\\)\\\", lines? \\([0-9]*\\)[:]` 1 2)

;; MIPS RISC CC - the one distributed with Ultrix:
;; ccom: Error: foo.c, line 2: syntax error
("rror: \\([^\\"\\n\\t]+\\), line \\([0-9]*\\)` 1 2)

;; IBM AIX PS/2 C version 1.1:
;; ***** Error number 140 in line 8 of file errors.c *****
("in line \\([0-9]*\\) of file \\([^\\"\\n]+[^.\\"n]\\)\\,.?` 2 1)

```

这个列表里的元素都是由三个部分组成的：一个正则表达式和两个数字。正则表达式

式的作用是按特定编译器所使用的格式对出错信息进行匹配，它被归组为几个子表达式。第一个数字告诉 Emacs 出错信息里的文件名被保存在第几个子表达式里；第二个数字则用来指明行号保存在第几个子表达式里。

比如说，列表的最后一项包含着正则表达式：

```
*in line \\([0-9]+\\) of file \\([^\n]+[^.\n]\\)\\..?*
```

然后是数字 2 和 1，这就意味着被归为第二组的子表达式包含着文件名，被归为第一组的子表达式包含着行号。如果你使用的是这个编译器（运行着 IBM 版 UNIX 操作系统的 Intel x86 个人电脑上的 C 语言编译器），你看到的出错信息就将是下面这样的：

```
***** Error number 140 in line 8 of file errors.c *****
```

正则表达式对出错信息中“in line”之前的内容不做处理。接下来，它找到了与第一个子表达式 “[0-9]+”（一个或更多个数字）相匹配的行号“8”，以及与第二个子表达式 “[^\n]+[^.\n]”（一个或更多个不是空格也不是换行符的字符，后面跟着一个不是英文句号，不是空格，也不是换行符的单个字符）相匹配的文件名“errors.c”。

现在可以着手解决“情况 5”中的问题了。为了使 Emacs 能够对新 C++ 编译器给出的德文出错信息进行分析，我们决定给 compilation-error-regexp-alist 列表增加一个元素。新编译器给出的出错信息采用的是下面这样的格式：

```
Fehler auf Zeile linenum in filename: text of error message
```

下面是我们准备添加到 **compilation-error-regexp-alist** 里去的元素：

```
(*Fehler auf Zeile \\([0-9]+\\) in\\([^\t]+\\):\\..?*)
```

注意观察编译器报告的出错信息，我们发现它和刚才那个例子一样：被归为第二组的子表达式将负责匹配文件名，被归为第一组的子表达式负责匹配行号。

为了把这个元素添加到变量 **compilation-error-regexp-alist** 里，我们需要把下面这些语句添加到 “.emacs” 文件里：

```
(setq compilation-error-regexp-alist  
      (cons '(*Fehler auf Zeile \\([0-9]+\\) in\\([^\t]+\\):\\..?*)  
            compilation-error-regexp-alist))
```

这个例子与我们往变量 **auto-mode-alist** 里添加一种新的语言编辑模式的支持(参见上一章内容)有很多相似之处。

正则表达式操作符汇总

表13-6为我们对正则表达式操作符的讨论画上了句号,我们在这一章讲过的所有操作符都列在这个表格里。

表 13-6: 正则表达式操作符速查表

操作符	作用
.	匹配任意一个字符
*	匹配其前面的字符或字符组出现零次或更多次的情况
+	匹配其前面的字符或字符组出现一次或更多次的情况
?	匹配其前面的字符或字符组出现零次或一次的情况
[...]	字符的集合, 参见下面的说明
\(开始一个字符组
\)	结束一个字符组
\!	匹配 “\\!” 前、后的子表达式
^	如果出现在正则表达式的开始, 匹配文本行首或字符串的开始
\$	如果出现在正则表达式的末尾, 匹配文本行尾或字符串的末尾
\n	在正则表达式里匹配换行符 LINEFEED
\t	在正则表达式里匹配制表符 TAB
\<	匹配单词的开始
\>	匹配单词的结尾

下面的操作符出现在字符集里时的含义:

^	如果出现在字符集的开始, 则该字符集将被用做不匹配字符集
-	用来设定字符范围

下面的操作符出现在 *regexp* 替换字符串里时的含义:

\N	替换第 N 个 “\\(” 和 “\\)” 之间的部分匹配, 从左归组操作符 “\\(” 向右编号, 编号从 1 开始
----	-------------------------------------------------------------

最后, 如果用两个反斜线字符对以下字符进行转义, 它们将转变为正则表达式的操

作符（我们没有对这些操作符进行介绍）：“b”、“B”、“w”、“W”、“s”、“S”、“`”（单引号）和“`”（反引号）。请注意这些“反斜线字符转义陷阱”。

使用正则表达式的函数

使用正则表达式的用户级命令有 **re-search-forward**、**re-search-backward**、**replace-regexp**、**query-replace-regexp**、**isearch-forward-regexp** 和 **isearch-backward-regexp**，它们也都能用来编写 LISP 代码（虽然在 LISP 代码里使用递增查找是难以想像的事）。本章后面关于定制主编辑模式的内容里，有一个使用 **re-search-forward** 函数的例子。

还有一些函数虽然也使用正则表达式，但不能做为用户级命令来使用。这些函数里最常用的大概是 **looking-at**。这个函数的输入参数是一个正则表达式，它的作用是：如果光标后面的文本匹配上该正则表达式，返回“t”（否则返回“nil”）；如果真的有一个匹配，它会截取“\\(”和“\\)”之间的片段并保存起来供今后使用，就像我们刚才见过的那样。**string-match** 函数也有类似的行为：它需要两个参数，一个正则表达式和一个字符串。如果字符串的某个片段能够与正则表达式匹配，**string-match** 函数将返回匹配片段的起始位置在该字符串里的索引；如果没有匹配，则返回“nil”。

match-beginning 和 **match-end** 函数可以用来检索被保存起来的字符串匹配片段。它们以正则表达式中的子表达式的编号（比如 **replace-regexp** 命令的替换字符串里的“\\N”）为输入参数，返回的是字符串匹配片段的起始点（**match-beginning**）或结束点（**match-end**）在编辑缓冲区里的字符位置。如果输入参数为数字“0”，就把用正则表达式匹配到的整个字符串的起始点，和结束点的字符位置做为返回值。

上面介绍的这些函数还需要另外两个函数的帮助才能真正起作用：我们必须知道怎样才能把编辑缓冲区里的文本转换为一个字符串。没问题：**buffer-string** 能够把整个编辑缓冲区做为一个字符串返回；**buffer-substring** 利用两个整数参数（用来给出子字符串在编辑缓冲区里的起始位置和结束位置）把想截取的子字符串返回。

有了这些函数，我们就能用下面这样的 LISP 代码把正则表达式的第 N 个子表达式，在编辑缓冲区里匹配到的片段放在一个字符串里返回：



```
(buffer-substring (match-beginning N)
                  (match-end N))
```

需要使用这种结构的地方实在是太多了，所以 Emacs 特意准备了一个内部函数 **match-string** 作为这个结构的快捷方式：**(match-string N)** 的返回值与上面这条 LISP 语句的完全一样。

为了把这个函数的工作情况讲清楚，我们一起来看一个例子。接着刚才编写用来分析编译器出错信息的 LISP 代码的例子往下说。你的代码依次使用 **compilation-error-regexp-alist** 列表的每一个元素，来检查编辑缓冲区中的文本是否与其中的某条正则表达式匹配。如果出现了匹配，你的代码将把文件名和行号提取出来，然后打开相应的文件，再到达指定的行号。

遍历列表中每一个元素的代码超出了我们目前的学习范围，但不要紧。这个例程的基本框架是下面这样的：

```
for each element in compilation-error-regexp-alist
  (let ((regexp the regexp in the element)
        (file-subexp the number of the filename subexpression)
        (line-subexp the number of the line number subexpression))
    (if (looking-at regexp)
        (let ((filename (match-string file-subexp))
              (linenum (match-string line-subexp)))
          (find-file-other-window filename)
          (goto-line linenum)
        (otherwise, try the next element in the list)))
```

第二条 **let** 语句通过编号为 **file-subexp** 的子表达式，把文件名从出错信息所在的编辑缓冲区里提取出来；然后再用同样的方法根据编号为 **line-subexp** 的子表达式提取出行号（然后把行号从字符串转换为一个数字）。接下来，这段代码打开相应的文件（在另外一个窗口里，不是出错信息编辑缓冲区所在的窗口）并到达出现错误的行号。

本章后半部分内容里用来实现计算器模式的代码，包含另外几个使用 **looking-at**、**match-beginning** 和 **match-end** 函数的例子。

找出其他的内部函数

Emacs 包含数以百计的内部函数，它们都可以用在编写的 LISP 代码当中。要想找出哪个函数最适合用来完成某项工作也不是很困难。

需要弄清楚的第一件事是：最常用的函数差不多都能通过键盘命令来访问。如果想使用某个函数，可以通过“**C-h k**”（命令名是 **describe-key**）命令查出其函数名（参见第十六章）。用这个方法可以查到待查命令的完整文档；与此相对的是“**C-h c**”（命令名是 **describe-key-briefly**）命令，它只给出待查命令的名字。请注意，在某些场合，一些常见的键盘命令用做LISP函数时需要带参数。**forward-word** 就是一个例子；为了得到与按下“**ESC f**”组合键同样的效果，必须使用“(**forward-word 1**)”。

为某种操作寻找适用函数的另一个强大工具是 **command-apropos** (“**C-h a**”组合键) 帮助函数。这个帮助函数能够根据给定的正则表达式，把所有与之匹配的命令都查出来，然后把这些命令的按键绑定（如果有）和文档显示在一个“help”窗口里。它特别适合用来寻找一个能够完成某项“基本”工作的命令。例如，如果想知道哪些命令是对单词进行操作的，请按下“**C-h a**”组合键，再输入单词“*word*”，几十个与单词有关的编辑命令和它们的文档就会显示出来。

但 **command-apropos** 命令只能查出用做键盘命令的函数的资料。**apropos** 命令的功能更强，但它没有被绑定为一个帮助热键（只能用输入“**ESC x apropos RETURN**”的方法来使用它）。给定一个正则表达式，**apropos** 将把与之匹配的各种函数、变量以及其他符号的资料都显示出来。但我们也必须提醒大家：如果让 **apropos** 查找的是一个比较基本的概念，那么它可能会执行很长的时间，而且会生成一个非常长的清单。

查找能力最强的命令是 **super-apropos**，它也没有被绑定到任何按键上。**super-apropos** 命令不仅具备 **apropos** 命令所具备的功能，还能对函数、变量或其他符号的文档字符串 (documentation strings) 进行搜索，把与给定正则表达式匹配的东西都找出来。当然，如果不注意节制，**super-apropos** 命令的执行时间要比 **apropos** 命令更长（而且会产生长度令人难以置信的输出）。

如果需要使用 **apropos** 命令来查找关于函数的资料，千万要把选好关键字。不过，需要 **apropos** 出马的场合并不是很多。这是因为如果想找的函数是比较常用或比较基本的，那么多半 Emacs 已经内置，因而也就用不着使用 **apropos** 命令了。

即使找到了感兴趣的函数，**apropos** 或 **super-apropos** 命令给出的资料也有可能不足以让你彻底了解该函数的功能、参数情况、使用方法或者其他事项。此时，最好

的解决方法就是直接研读 Emacs 的 LISP 源代码，从中找出那个函数的使用示例。Emacs 的 LISP 源代码一般保存在各位系统中一个名为 “*emacs-source/lisp*” 的子目录里，其中的 “*emacs-source*” 是系统上 Emacs 源代码的根目录名称。先用 **grep** 或其他查找工具查出需要的函数示例保存在哪个文件里，再打开那个文件看看函数周围的上下文情况。虽说 Emacs 的内部 LISP 源代码中的文档注释并不是很充足，但它提供的函数用法示例还是很有帮助意义的——甚至能对大家自己编写的函数有所启发。

以上介绍的各种 Emacs LISP 基本概念和知识已经足以让大家编写出很多有用的 Emacs 命令了。在我们分析过的函数示例中，既有 LISP 语言本身的基础函数，也有 Emacs 的内部函数。各位已经具备了运用在这一章里学习的概念和技巧解决其他问题的能力。换句话说、各位已经踏上了成为一名优秀 Emacs LISP 程序员的道路。为了检查自己的学习情况，希望大家能够在 **count-words-buffer** 函数代码的基础上写出下面这几个函数：

count-lines-buffer	统计出编辑缓冲区里有多少个文本行
count-words-region	统计出文本块里有多少个单词
what-line	给出光标当前位置的行号

主编辑模式程序设计实例

对 Emacs LISP 程序设计比较熟悉之后，你可能发现自己想用一种主编辑模式让 Emacs 做一点“额外工作”。在以前各章里，我们对文本输入、文字排版和程序设计语言等工作所使用的主编辑模式进行了探讨。这些主编辑模式大都相当复杂，普通程序员很难写出这样的程序来。因此，我们将提供一个简单的主编辑模式程序设计实例，希望大家能够从中学习到编写自己的主编辑模式所需的概念。然后，我们将在下一节介绍，如何在不对原来实现现有主编辑模式的有关 LISP 代码做任何改动的前提下对它们进行定制。

我们准备开发的 **calc-mode** 主编辑模式，将实现一个与 UNIX 的 **dc** (desk calculator，桌面计算器) 命令功能近似的计算器。它是一个使用逆波兰表示法（一种基于堆栈的表示法）的计算器，这种表示法因惠普公司 (Hewlett Packard) 的推广而变得很流行。在介绍完主编辑模式的几个重要组件，和计算器模式几个有趣的功能之后，我们将把这个主编辑模式完整的 LISP 代码呈现给大家。

主编辑模式的组件

就一个主编辑模式而言，它是由各种把它集成到 Emacs 里的组件构成的。这些组件包括：

- 符号——即用来实现这个主编辑模式的函数的名称。
- 名字——这个主编辑模式准备显示在状态行括号里的名称。
- 局部键位图——用来定义这个主编辑模式里各种命令的按键绑定。
- 变量和常数——为了实现这个主编辑模式，LISP 代码需要使用一些局部变量和局部常数。
- 专用编辑缓冲区——这个主编辑模式下有特殊用途的编辑缓冲区。

我们来一个一个地解决这些问题。先设置主编辑模式的符号，方法是把用来实现这个主编辑模式的函数的名字赋值给全局变量 **major-mode**，如下所示：

```
(setq major-mode 'calc-mode)
```

类似地，设置主编辑模式的名字，需要把相应的字符串赋值给全局变量 **mode-name**，如下所示：

```
(setq mode-name "Calculator")
```

局部键位映射图需要用第十一章介绍的函数来定义。因为我们的计算器模式只需绑定一个按键（用来输入换行符 LINEFEED 的“C-j”组合键），所以我们决定使用 **make-keymap** 命令一个特殊形式——如果需要绑定的按键数量不大，使用 **make-sparse-keymap** 命令将更有效。为了把一个键位映射图设置为某主编辑模式的局部键位映射图，我们需要调用函数 **use-local-map**，如下所示：

```
(use-local-map calc-mode-map)
```

我们已经见过几种定义变量的方法，比如用 **setq** 给它们赋一个值，或者用 **let** 来定义函数中的局部变量等。更正规的变量定义方法是使用 **defvar** 函数，它允许程序员把变量的文档资料集成到“C-h v”组合键（命令名是 **describe-variable**）等在线帮助功能里。这个函数的语法格式是：

```
(defvar varname initial-value "description of the variable")
```

defvar 函数还有一个变体叫做 **defconst**，可以用它来定义常数（即不会发生变化的值）。请看下面的例子：

```
(defconst calc-operator-regexp "[+\-*%]"  
  "Regular expression for recognizing operators.")
```

这条语句定义了一个用来查找算术运算符的正则表达式。正像大家将会看到的那样，我们将给为计算器模式所定义的一切函数、变量和常数都加上一个“calc-”前缀。这是主编辑模式普遍采用的习惯做法；比如说，C++ 模式里的一切名字就都是以“c++-”开头的。这种做法是一个很好的习惯，它能帮助避免使用的名字与 Emacs 里数以千计的其他函数、变量或者诸如此类的东西发生重名冲突。

接下来的是把变量设置为主编辑模式的局部变量，这样，运行在某主编辑模式下的编辑缓冲区才会知道有哪些东西可用（注6）。设置局部变量的工作要用**make-local-variable** 函数来完成，如下所示：

```
(make-local-variable 'calc-stack)
```

请注意，这里使用的是变量的名字而不是它的值；所以需要在变量名的前面加上一个单引号，使它成为一个符号。

最后，各种主编辑模式会用到一些特殊的编辑缓冲区，这些编辑缓冲区与任何文件都没有关联关系。比如，“C-x C-b”（命令名是**list-buffers**）命令会创建一个名为“*Buffer List*”的编辑缓冲区。要想在新窗口里创建一个编辑缓冲区，需要使用**pop-to-buffer** 函数，如下所示：

```
(pop-to-buffer "*Calc")
```

pop-to-buffer 函数还有几个很有用的变体。我们这里的主编辑模式程序设计实例没有使用它们，但它们在其他场合是很有用的：

switch-to-buffer 与第四章介绍的“C-x b”命令作用相同，也可以用来编写 LISP 代码，但需要给出一个编辑缓冲区名字做为参数。

注 6： 遗憾的是，因为这些变量是先进行定义，然后才被设置为主编辑模式的局部变量，所以与全局变量产生重名冲突的可能性是存在的。因此，给变量起名字时一定不要用全局变量已经使用的名字，这一点非常重要。避免出现重名冲突的应对策略之一就是让变量名以主编辑模式的名字开头。

set-buffer 只能用在 LISP 代码里，作用是将编辑缓冲区指定为文本编辑工作使用；最适合用来在 LISP 函数里创建一个临时性的“工作”编辑缓冲区。

列表：LISP 语言的基本概念

采用逆波兰表示法的计算器使用一种名为“堆栈”的数据结构。可以把堆栈想像为咖啡店里摞在一起的许多碟子。向采用逆波兰表示法的计算器输入一个数字时，等于是把这个数字“压入”堆栈。输入一个运算符（比如加号或减号）时，等于是把堆栈最顶部的两个数字“弹出”堆栈，对它们进行加法或者减法操作，然后再把计算结果压入堆栈。

列表是 LISP 语言中的一个基本概念，它充分体现了堆栈结构本质特征。列表是把 LISP 与其他程序设计语言区分开来的主要概念。这种数据结构由两个部分组成：头元素和尾元素。纯粹出于历史沿革方面的原因，它们在 LISP 语言里又被分别称为 **car** 和 **cdr**。可以把这两个术语看做是“列表里的第一样东西”和“列表里的其余东西”。如果给 **car** 和 **cdr** 函数提供一个列表做为参数，它们就将分别返回该列表的头元素和尾元素（注 7）。人们经常用两个函数来构造列表。**cons**（构造）函数带两个输入参数，而这两个参数将分别被设置为列表的头元素和尾元素。**list** 函数以多个数据元素为输入参数，用它们构造出一个列表。请看下面的例子：

```
(list 2 3 4 5)
```

上面这个函数将构造出一个由从 2 到 5 的四个数字组成的列表，而

```
(cons 1 (list 2 3 4 5))
```

将构造出一个由从 1 到 5 的五个数字组成的列表。如果以这个列表为输入参数，**car** 函数将返回 “1”，**cdr** 函数将返回列表 “(2 3 4 5)”。

这些概念是非常重要的，因为堆栈（我们的计算器模式就使用了堆栈概念）可以很容易地被实现为列表。为了把值 “x” 压入堆栈 **calc-stack**，我们只需写出下面这条语句：

注 7： 有经验的 LISP 程序员会注意到 Emacs LISP 没有提供 **cadr** 和 **cdar** 等标准的列表操作函数。

```
(setq calc-stack (cons x calc-stack))
```

如果我们想取出堆栈最顶部的值，就可以用下面这条语句得到它：

```
(car calc-stack)
```

如果想把最顶部的值弹出堆栈，我们只需写出：

```
(setq calc-stack (cdr calc-stack))
```

请记住，列表的元素可以是包括其他列表在内的任何东西。这就是人们把列表称为一种“递归”数据结构的原因。事实上，对LISP语言里的任何一样东西来说，如果它不是一个原子项，就会是一个列表。这里面也包括函数，它可以看做一个由函数名、参数、准备求值的表达式等元素组成的列表。把函数看做是一种列表的观点很快就会显出它的优越性来。

程序设计示例：计算器模式

我们把用来实现计算器模式的完整的LISP代码列在这一小节的最后部分；大家在阅读下面的解释内容时可以随时去查阅它们。如果把那些代码都输入或扫描到机器里，就可以通过输入“**ESC x calc-mode RETURN**”命令的方法来使用这个计算器。光标将移动到“*Calc*”编辑缓冲区里。可以输入一行数字和运算符，按下“**C-j**”组合键的时候，计算器会对这一行进行求值。我们的计算器有3个特殊的命令，它们是：

- = 显示堆栈最顶部的那个值
- p 显示整个堆栈的内容
- c 清除堆栈

数字之间用空格分隔，而运算符与数字之间则可以没有空格。请看下面的例子。如果输入：

```
4 17*6=
```

再按下“**C-j**”组合键，我们的计算器就会开始计算(4*17)-6，并把结果62显示出来。

计算器模式的核心代码是函数 **calc-eval** 和 **calc-next-token**。（从本节结尾部分里的源代码可以更清楚地看出这一点。）**calc-eval** 函数被绑定到计算器模式下的“C-j”组合键上。计算工作将从“C-j”之前的行首开始进行，它将调用 **calc-next-token** 函数，依次取出该行中的每一个记号（数字、运算符或者命令字母）并对它进行求值。

calc-next-token 函数使用一个 **cond** 结构，这个 **cond** 结构将根据正则表达式 **calc-number-regexp**、**calc-operator-regexp** 和 **calc-command-regexp** 来检查光标位置上是否有一个数字、运算符或者命令字母。根据得到匹配的是哪一个正则表达式，它将把变量 **calc-proc-fun** 设置为应该被调用执行的函数 (**calc-push-number**、**calc-operate** 或 **calc-command**) 的名字 (符号)，并且会把变量 **tok** 设置为正则表达式匹配的结果。

再说说 **calc-eval** 函数，它体现了把函数看做是一种列表的观念。**funcall** 函数把 LISP 语言中代码和数据之间几乎没有区别的特点显露无余。我们可以用一个符号和一组表达式构造出一个列表，然后再把这个列表当做一个函数来进行求值。那个符号就是函数名，而那些表达式就是这个函数的参数；**funcall** 正是这样做的。请看：

```
(funcall calc-proc-fun tok)
```

这条语句把 **calc-proc-fun** 的符号值看做将被调用的函数的名字，然后以 **tok** 做为其参数进行调用。接下来，这个函数将进行下面这 3 种操作中的一种：

- 如果记号是一个数字，**calc-push-number** 将把这个数字压入堆栈。
- 如果记号是一个运算符，**calc-operate** 将对堆栈最顶部的两个数字进行相应的操作（参见下面的解释）。
- 如果记号是一个命令字母，**calc-command** 将执行相应的命令动作。

calc-operate 函数把函数也是数据列表的概念又往前推进了一步，它把用户输入的记号直接转换为一个函数（对计算器来说，就是某个算术运算符）。这项工作是由 **read** 函数完成的，这个函数以一个字符串为输入参数并把它转换为一个符号。这样，函数 **funcall** 和 **read** 在 **calc-operate** 函数里的合作情况如下所示：

```
(defun calc-operate (tok)
  (let ((op1 (calc-pop)))
```

```
(op2 (calc-pop))
(calc-push (funcall (read tok) op2 op1)))
```

这个函数以一个算术运算符的名字（当做一个字符串）做为自己的输入参数。从前面的讲解里我们已经知道，字符串 `tok` 是从“*calc*”编辑缓冲区提取出来的一个记号；对我们的计算器来说，这个记号就是一个算术运算符，比如“+”或“*”。`calc-operate` 函数通过 `pop` 函数把堆栈最顶部的两个数字弹出堆栈，这与前面介绍的 `cdr` 函数的工作情况有些相似。`read` 函数把记号转换为一个符号，即某个算术函数的名字。假设这个运算符是“+”，则 `funcall` 的调用情况就将是下面这个样子：

```
(funcall '+ op2 op1)
```

这相当于以两个参数来调用加法函数，这与“正宗”的“(+ op2 op1)”效果完全相同。最后，加法函数的计算结果被压入堆栈的最顶部。

这套把戏是很有必要的，它使用户输入的算术运算符（比如加号“+”）被 LISP 自动转换为一个加法函数。我们也可以不那么精细，当然，也就不那么有效率；比如说，我们完全可以用一个 `cond` 结构写出 `calc-operate` 函数，就像下面这样：

```
(defun calc-operate (tok)
  (let ((op1 (calc-pop))
        (op2 (calc-pop)))
    (cond ((equal tok "+")
           (+ op2 op1))
          ((equal tok "-")
           (- op2 op1))
          ((equal tok "*")
           (* op2 op1))
          ((equal tok "/")
           (/ op2 op1)))
          (t
           (% op2 op1)))))
```

最后，请记住计算器模式的代码是函数 `calc-mode`，我们编写的这个主编辑模式将由它来启动。它先创建（并弹出）“*Calc*”编辑缓冲区。然后，它会清除该编辑缓冲区中已经存在的所有局部变量，把堆栈初始化为“nil”，再创建一个局部变量 `calc-proc-fun`（参见前面的讨论）。最后，它把 `calc-mode` 设置为主编辑模式，设置好这个主编辑模式的名称，并启用计算器模式的局部键位图。



计算器模式的 LISP 代码

现在，相信大家已经能够看懂计算器模式的全部代码了。你会发现代码根本不算多！这要完全归功于强大的 LISP 语言和多功能的 Emacs 内部函数。一旦掌握了这些代码的工作原理，你就可以开始编写自己的主编辑模式。不多说了，下面就是我们编写的计算器模式的代码：

```
; ; Calculator mode.  
; ;  
; ; Supports the operators +, -, *, /, and % (remainder).  
; ; Commands:  
; ; c      clear the stack  
; ; =      print the value at the top of the stack  
; ; p      print the entire stack contents  
;  
  
(defvar calc-mode-map nil  
  "Local keymap for calculator mode buffers.")  
  
; set up the calculator mode keymap with  
; C-j (linefeed) as 'eval' key  
(if calc-mode-map  
    nil  
    (setq calc-mode-map (make-sparse-keymap))  
    (define-key calc-mode-map 'C-j 'calc-eval))  
  
(defconst calc-number-regexp  
  '^-[0-9]+\\.?[0-9]*[0-9]*([e[0-9]+])?'  
  "Regular expression for recognizing numbers.")  
  
(defconst calc-operator-regexp '[-+*/%]'  
  "Regular expression for recognizing operators.")  
  
(defconst calc-command-regexp "[c=ps]"  
  "Regular expression for recognizing commands.")  
  
(defconst calc-whitespace '[ \t]'  
  "Regular expression for recognizing whitespace.")  
  
;; beep and print an error message  
(defun calc-error-message (str)  
  (beep)  
  (message str)  
  nil)  
  
;; stack functions  
(defun calc-push (num)  
  (if (numberp num)  
      (setq calc-stack (cons num calc-stack))))
```

```
(defun calc-top ()
  (if (not calc-stack)
      (error-message 'stack empty.)
      (car calc-stack)))

(defun calc-pop ()
  (let ((val (calc-top)))
    (if val
        (setq calc-stack (cdr calc-stack))
        val))

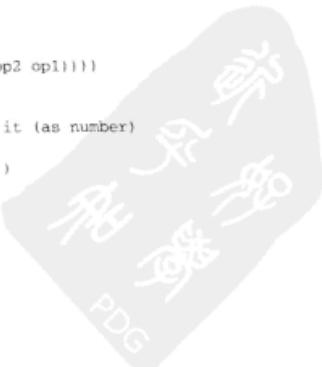
;; functions for user commands:
(defun calc-print-stack ()
  "Print entire contents of stack, from top to bottom."
  (if calc-stack
      (progn
        (insert "\n")
        (let ((stk calc-stack))
          (while calc-stack
            (insert (number-to-string (calc-pop)) " "))
          (setq calc-stack stk)))
        (error-message 'stack empty.)))

(defun calc-clear-stack ()
  "Clear the stack."
  (setq calc-stack nil)
  (message "stack cleared."))

(defun calc-command (tok)
  "Given a command token, perform the appropriate action."
  (cond ((equal tok 'c)
         (calc-clear-stack))
        ((equal tok "-")
         (insert "\n" (number-to-string (calc-top))))
        ((equal tok 'p)
         (calc-print-stack))
        (t
         (message (concat "invalid command: " tok)))))

(defun calc-operate (tok)
  "Given an arithmetic operator (as string), pop two numbers
off the stack, perform operation tok (given as string), push
the result onto the stack."
  (let ((op1 (calc-pop))
        (op2 (calc-pop)))
    (calc-push (funcall (read tok) op2 op1)))

(defun calc-push-number (tok)
  "Given a number (as string), push it (as number)
onto the stack."
  (calc-push (string-to-number tok)))
```



```

(defun calc-invalid-tok (tok)
  (error-message (concat "Invalid token: ` " tok))
  (sit-for 0)) ;make sure message is displayed

(defun calc-next-token ()
  "Pick up the next token, based on regexp search.
As side effects, advance point one past the token,
and set name of function to use to process the token."
  (let (tok)
    (cond ((looking-at calc-number-regexp)
           (goto-char (match-end 0))
           (setq calc-proc-fun 'calc-push-number))
          ((looking-at calc-operator-regexp)
           (forward-char 1)
           (setq calc-proc-fun 'calc-operate))
          ((looking-at calc-command-regexp)
           (forward-char 1)
           (setq calc-proc-fun 'calc-command)))
      (t
       (setq calc-proc-fun 'calc-invalid-tok)))
    ;; pick up token and advance past it (and past whitespace)
    (setq tok (buffer-substring (match-beginning 0) (point)))
    (if (looking-at calc-whitespace)
        (goto-char (match-end 0)))
    tok))

(defun calc-eval ()
  "Main evaluation function for calculator mode.
Process all tokens on an input line."
  (interactive)
  (beginning-of-line)
  (while (not (eolp))
    (let ((tok (calc-next-token)))
      (funcall calc-proc-fun tok)))
  (insert "\n"))

(defun calc-mode ()
  "Calculator mode, using H-P style postfix notation.
Understands the arithmetic operators +, -, *, / and %,
plus the following commands:
  c  clear stack
  =  print top of stack
  p  print entire stack contents (top to bottom)
Linefeed (C-j) is bound to an evaluation function that
will evaluate everything on the current line. No
whitespace is necessary, except to separate numbers."
  (interactive)
  (pop-to-buffer "*Calc* nil)
  (kill-all-local-variables)
  (make-local-variable 'calc-stack)
  (setq calc-stack nil)
  (make-local-variable 'calc-proc-fun)

```



```
(setq major-mode 'calc-mode)
(setq mode-name "Calculator")
(use-local-map calc-mode-map))
```

下面是计算器模式还值得改进或扩展的一些方面，我们把它们作为练习留给大家。这些练习不仅能够加深你对计算器模式的代码的理解，还能全面提高你的 Emacs LISP 程序设计水平。

- 增加一个求幂运算符 “`^`”（比如 `45^` 等于 `1024`）。Emacs LISP 没有提供内建的求幂函数，但可以利用内建函数 `expt` 来做这个练习。
- 让此计算器能够支持八进制和/或十六进制数字。八进制数字用一个“`0`”开头，十六进制数字用一个“`0x`”开头；这样，`017` 等于十进制的 `15`，而 `0x17` 等于十进制的 `23`。
- 增加两个运算符 “`\+`” 和 “`*`”，让它们对堆栈里的数字进行连加或者连乘；而不仅仅是对最顶部的两个数字进行运算（比如，“`4 5 6 \+`” 等于 `15`，而 “`4 5 6 *`” 等于 `120`）（注 8）。
- 做为考察大家 LISP 知识水平的附加题，请解决本章开始部分给出的“情况 5”里的问题，即在 Emacs 变量 `compilation-error-regexp-alist` 里为编译器出错信息找出一个匹配来。（提示：先给这个列表复制一个副本，然后反复弹出最顶部的元素，直到找到一个匹配或者列表已经穷尽。）

对现有编辑模式进行定制

在掌握了主编辑模式的程序设计方法之后，你可能觉得对现有的主编辑模式进行定制更方便些。很幸运，对大多数定制情况来说，根本不用修改任何主编辑模式现有的 LISP 代码；甚至无需查看那些代码。Emacs 的各种主编辑模式都有一些可以把你编写的代码添加到它上面去的“挂钩”。它们的名字恰好叫做“**mode-hooks**”（英文“hook”的意思就是“挂钩”）。Emacs 内建的各主编辑模式都有一个名为“**mode-name-hook**”的编辑模式挂钩，其中的“`mode-name`”是该编辑模式的名称或启动它的函数的名字。比如说，C 模式就有一个 `c-mode-hook`，而 shell 模式则有一个 `shell-mode-hook` 等等。

注 8：APL 程序员会看出这两个运算符是 APL 语言“扫描”操作符的变体。

那么，这个挂钩到底是什么呢？它是一种特殊的变量，这些变量的值是一些LISP代码，这些代码会在相应的编辑模式启动时得到运行。启动一个编辑模式就等于运行了一个LISP函数，这个函数通常要完成很多事情（比如为特殊命令设置按键绑定、创建编辑缓冲区和局部变量等等）；如果编辑模式有挂钩，那么负责启动编辑模式的函数所做的最后一件事情通常就是运行这些挂钩。总而言之，挂钩是负责实现某个编辑模式的LISP代码留出的一个“位置”，使你有机会改变它已经设置好的一些东西。比如说，你定义的按键绑定就将取代编辑模式的缺省绑定设置。

通过以前的学习，我们知道LISP代码可以是一个LISP变量的值，这一特点特别适合创建挂钩。在介绍如何创建一个挂钩的具体操作之前，我们先要介绍另外一个LISP基本函数：**lambda**。**lambda**也可以用来定义函数，在这一点上它与**defun**很相似；用LISP的术语来说，它们之间的区别是**lambda**定义的函数没有名字，就是用**lambda**定义的函数都是些“匿名函数”。**lambda**的语法格式如下所示：

```
(lambda (args)
  code)
```

其中的*args*是该函数的参数，而*code*是该函数的函数体。如果想把一个用**lambda**定义的函数作为一个值赋给某个变量，就必须对它进行“转义引用”以防止它被求值（即防止它被调用运行）。这就是说，必须使用下面这样的语句来进行赋值：

```
(setq var-name
  `(lambda ()
    code))
```

于是，在编写编辑模式挂钩的代码时，可以使用下面的格式：

```
(setq mode-name-hook
  `(lambda ()
    code for mode hook))
```

不过，你想定制的编辑模式可能已经有定义好的挂钩了。如果使用**setq**来定义你自己的挂钩，就会覆盖掉原有的挂钩。要想避免出现这样的问题，就应该改用**add-hook**函数：

```
(add-hook 'mode-name-hook
  '(lambda ()
    code for mode hook))
```

编辑模式挂钩最常见的用途是为某个编辑模式的特殊命令修改一个或者多个按键绑

定。先来看一个例子：我们在第八章介绍了一种可以用来绘制简单线条图形的工具—图形模式。图形模式里有几个用来设置缺省绘制方向的命令。把缺省绘制方向设置为“下”的命令是 **picture-movement-down**，它被绑定到“C-c .”（“C-c”后面跟着一个英文句号）组合键上。与对应 **picture-movement-left** 命令的“C-c <”组合键和对应 **picture-movement-up** 命令的“C-c ^”组合键相比，“C-c .”组合键不太容易记忆；所以我们现在想把 **picture-movement-down** 命令绑定到“C-c v”组合键上。图形模式所使用的键位图叫做 **picture-mode-map**，所以用来设置这个按键绑定的代码是：

```
(define-key picture-mode-map "\C-cv" 'picture-movement-down)
```

图形模式的挂钩叫做 **edit-picture-hook**（因为启动图形模式的命令是 **edit-picture**）。于是，为了把这个代码挂到图形模式的挂钩上去，就必须把下面这条语句添加到 “.emacs” 文件里：

```
(add-hook 'edit-picture-hook
          '(lambda ()
            (define-key picture-mode-map "\C-cv" 'picture-movement-down)))
```

这条指令创建了一个 **lambda** 函数，而它的函数体是一个按键绑定命令。以后，只要进入图形模式（从下次启动 Emacs 开始），这个绑定就会生效。

再来看一个稍微复杂点的例子，假设你正使用 **nroff** 或 **troff** 对文本进行排版。你进入了 **nroff** 模式（参见第九章），但发现 Emacs 没有提供用来输入标准的 **nroff / troff** 字体设置命令（即分别对应于粗体、斜体和罗马体（即“正常”体）的“\fB”、“\fI”和“\fR”）的编辑命令。于是，你决定自行编写一个函数来插入这几个字符串，并且想把它们绑定到 **nroff** 模式下的几个键盘操作命令上。

为了实现你的想法，你先得写出一个用来插入字体设置命令字符串的函数。这个函数还是比较简单的，如下所示：

```
(defun nroff-insert-bold ()
  (interactive)
  (insert "\fB"))

(defun nroff-insert-italic ()
  (interactive)
  (insert "\fI"))

(defun nroff-insert-roman ()
  (interactive)
  (insert "\fR"))
```

```
(interactive)
(insert "\\"ER")
```

请注意，作为 `insert` 函数的参数的字符串里必须加上双反斜线字符，这样才能让 Emacs 把它们解释为单个的反斜线字符；对“`(interactive)`”函数的调用也是必不可少的，这样 Emacs 才会把这些函数看做用户级命令。

下一个步骤是用第十一章介绍的技术，编写能把这几个函数绑定到 nroff 模式键位映射图中的按键动作代码；nroff 模式的键位图叫做 **nroff-mode-map**。假设你想把这几个函数分别绑定到“**C-c C-b**”（粗体）、“**C-c C-i**”（斜体）和“**C-c C-r**”（罗马体）组合键上。“**C-c**”是 Emacs 的许多内建编辑模式——比如我们在上一章里见到的各种语言编辑模式——所使用的前缀键。这段代码写起来也不困难，如下所示：

```
(define-key nroff-mode-map "\C-c\C-b" 'nroff-insert-bold)
(define-key nroff-mode-map "\C-c\C-i" 'nroff-insert-italic)
(define-key nroff-mode-map "\C-c\C-r" 'nroff-insert-roman)
```

最后，需要把这些 LISP 语句转换为 **nroff-mode-hook** 的一个值。下面是有关的代码：

```
(add-hook 'nroff-mode-hook
  '(lambda ()
    (define-key nroff-mode-map "\C-c\C-b" 'nroff-insert-bold)
    (define-key nroff-mode-map "\C-c\C-i" 'nroff-insert-italic)
    (define-key nroff-mode-map "\C-c\C-r" 'nroff-insert-roman)))
```

如果把上面这些函数的代码都添加到 “`.emacs`” 文件，以后再使用 nroff 模式的时候就会得到想要的功能。

下面是第三个例子。假设你正使用 C 语言进行程序设计，想用一个 LISP 函数来统计某个文件里有多少 C 语言的函数定义。下面那个函数将实现你的想法；它与我们在本章前面看到的 `count-lines-buffer` 例子有几分相似。这个函数将遍历当前编辑缓冲区，通过查找出现在行首的左花括号 “`(`” 来找出（和统计）其中到底有多少 C 语言的函数定义。如下所示：

```
(defun count-functions-buffer ()
  "Count the number of C function definitions in the buffer."
  (interactive)
  (save-excursion
    (goto-char (point-min))
    (let ((count 0))
      (while (re-search-forward "(" nil t)
        (setq count (1+ count)))
      (message "%d functions defined." count))))
```

这个函数的 **re-search-forward** 调用中额外多了两个参数：其第三个（即最后一个）参数的含义是“如果没有找到，就只返回 nil而不发出错误信息”。而它的第二个参数则必须设置为它的缺省值“nil”，这样才能引入第三个参数（注 9）。

接下来，我们想把这个函数绑定到 C 模式里的“C-c f”组合键上。下面是我们用来给 **c-mode-hook** 进行赋值的语句：

```
(add-hook 'c-mode-hook
  '(lambda ()
    (define-key c-mode-map "\C-cf" 'count-functions-buffer)))
```

把这两个函数定义添加到 “.emacs” 文件里，就可以在 C 模式里得到预期的功能。

做为编辑模式挂钩的最后一个例子，我们将完成在第十二章里许下的一个诺言。我们在讨论 C++ 模式时曾经介绍过两个可以用来替补 **forward-word** 和 **backward-word** 命令的编辑命令，它们分别是 **c-forward-into-nomenclature** 和 **c-backward-into-nomenclature** 命令；这两条替补命令可以把诸如“WordsLikeThis”之类的字符串当做三个而不是一个单词来对待，这项功能对 C++ 程序员来说是很有用的。摆在我们面前的问题是怎样才能让通常用来执行 **forward-word** 和 **backward-word** 命令的按键动作去执行这两个新命令。

最容易想到的答案是为 C++ 模式创建一个挂钩，把 **forward-word** 和 **backward-word** 命令的缺省绑定，即“**ESC f**”和“**ESC b**”组合键重新绑定到那两个新命令上，就像下面这样：

```
(add-hook 'c++-mode-hook
  '(lambda ()
    (define-key c++-mode-map "\e"
      'c-forward-into-nomenclature)
    (define-key c++-mode-map "\eb"
      'c-backward-into-nomenclature)))
```

（请注意，我们的按键绑定操作是在 C++ 模式的局部键位映射图 **c++-mode-map** 上进行的。）可遇到这两个组合键已被 Emacs 重新绑定过的情况该怎么办？遇到 **forward-word** 和 **backward-word** 命令被绑定到其他组合键上的情况（这种可能性

注 9： **re-search-forward** 函数以及其他查找函数的第二个参数是用来对该查找操作进行限制的：如果给它一个整数值 *N*，就意味着跳过 *N* 个字符位置不做查找。如果是缺省值“nil”，就意味着不对查找操作加以限制。

是存在的)又该怎么办?我们需要查明这两个函数到底被绑定在哪些组合键上,再把它们都重新绑定为新的函数。

幸好有一个不太常用的函数可以帮我们查出这些信息,它就是**where-is-internal**函数。这个函数实现求助命令**where-is**(我们将在第十六章对**where-is**命令进行介绍)的功能。**where-is-internal**函数的作用是返回一个由LISP原子项构成的列表,该列表的每个元素对应着一个组合键,而作为其输入参数的函数就是绑定在这些组合键上的。我们可以用一个**while**循环来遍历这个列表以完成必要的重新绑定操作。下面是有有关的代码:

```
(add-hook 'c++-mode-hook
  '(lambda ()
    (let ((fbinds (where-is-internal 'forward-word))
          (bbinds (where-is-internal 'backward-word)))
      (while fbinds
        (define-key c++-mode-map (car fbinds)
          'c-forward-into-nomenclature)
        (setq fbinds (cdr fbinds)))
      (while bbinds
        (define-key c++-mode-map (car bbinds)
          'c-backward-into-nomenclature)
        (setq bbinds (cdr bbinds)))))))
```

let语句的头两行代码把**forward-word**和**backward-word**命令的各种绑定都找出来并分别放到局部变量**fbinds**和**bbinds**里。

接下来是两个**while**循环,它们的各种情况与本章前面用来实现计算器模式的LISP代码中的**print-stack**函数很相似。采用**while**结构来遍历列表元素的这种方法在LISP程序设计工作中是很常见的:先对列表的第一个元素(即列表的头元素**car**)进行处理,然后用“(**setq list (cdr list)**)”语句把它从列表里删除;而整个循环将在列表变为空列表(**nil**)——即**while**语句的循环条件不成立时停止执行。

就上面这段代码而言,第一个**while**循环将逐个地取出**where-is-internal**函数找到的**forward-word**命令的每个绑定,并在C++模式局部键位图**c++-mode-map**中创建一个对应于新命令**c-forward-into-nomenclature**的绑定。第二个**while**循环将对**backward-word**命令和**c-backward-into-nomenclature**命令做同样的处理。

周围的代码将这些循环作为挂钩到安装C++模式,因此仅当C++模式被调用并在编辑缓冲区(也处于该模式)是活动的时,才能进行重新绑定。

大家可能已经注意到了，我们在以前各章里给出的编辑模式定制示例有的使用了挂钩，可有的却没有使用挂钩。比如说，上一章中用来设置 C 或 C++ 缩进样式的代码里就使用了一个挂钩：

```
(add-hook 'c-mode-hook
  '(lambda ()
    (c-set-style "stylename")
    (c-toggle-auto state)))
```

而用来给 **c-macro-expand** 命令另外设置一个 C 语言预处理器命令名的代码却没有使用挂钩：

```
(setq c-macro-preprocessor "/usr/local/lib/cpp -C")
```

为什么会有这样的呢？按照规定，定制任一编辑模式的正确做法是必须通过它的挂钩来进行——刚才的例子本应该用下面这条语句来完成：

```
(add-hook 'c-mode-hook
  '(lambda ()
    (setq c-macro-preprocessor "/usr/local/lib/cpp -C")))
```

如果只是给变量重新设置一个值，就可以不使用挂钩；但如果想运行函数，例如 **c-set-style** 或者那些用来绑定按键的函数，就必须使用挂钩。这种“两面派”做法的具体原因涉及到 LISP 语言深层次的设计理念，它的基本原理是这样的：

诸如 **c-macro-preprocessor** 之类的变量是属于编辑模式局部范畴之内的，如果没有启动对该变量做出定义的编辑模式，它们就不会存在。因此，如果不是正在编辑 C 或 C++ 代码，变量 **c-macro-preprocessor** 就不会存在于当前运行的 Emacs 中，因为还没有加载 C 模式（参见下面的说明）。但只要 *.emacs* 文件包含一条给这个变量赋值的 **setq** 语句，那么不管是否使用 C 模式，这个变量就已经存在了。Emacs 能够很好地应付这种情况：当加载 C 模式时，它会注意到已经设置过这个变量的值，于是就不会再覆盖掉它。

但函数的情况就不同了。如果 *.emacs* 文件里有一个属于某个编辑模式的局部函数（比如 **c-set-style**），那么 Emacs 多半会给出一条出错信息“Error in init file (初始化文件中发现错误)”，因为它此时根本不知道这个函数属于哪个编辑模式，更不用说它的作用。因此，必须把这个函数挂到 C 模式的一个挂钩上：等 Emacs 运行挂钩时，因为它已经加载 C 模式，所以就会知道这个函数的作用。



我们给出的这些关于挂钩的示例只是为了让大家了解在定制Emacs主编辑模式的道路上能够走多远。总而言之，因为有了挂钩，你能够在不触动实现编辑模式的LISP代码的前提下对它们进行各种各样的定制，其数量之多绝对超乎你的想像。

建立自己的LISP开发库

在做了大量的Emacs LISP程序开发工作之后，你可能想把自己写的LISP函数和程序包总结为一个开发库，这样，你就能在必要时按自己的意愿来调用它们。如果你只定义了几个“小”函数，把它们都放到`.emacs`文件里当然不会有问题；可如果你正在为比较专业的用途开发规模较大的代码，把它们都放到`“.emacs”`文件里就不是个好主意——你肯定不想让Emacs在你每次启动它的时候都因为要处理这些代码而花费大量的时间。解决这一问题的办法就是建立自己的LISP开发库，就像Emacs自带的*lisp*目录一样，Emacs的内建LISP代码都放在这个目录里。一旦你建立起自己的开发库，就能做到只在必要的时候加载必要的LISP程序包，不用再把它们全都塞到`“.emacs”`文件里。

只需两个简单的步骤就能创建一个开发库。首先，创建一个目录（用UNIX操作系统的`mkdir`命令），你的LISP代码就将存放在这个目录里。大多数人会在他们的主目录里创建出一个名为`“lisp”`的下级子目录。LISP文件的名字通常是以`“.el”`结尾的（你的`“.emacs”`文件是个例外）。第二步是让Emacs知道有这么一个目录，这样，当你试图加载某个LISP程序包时，Emacs才会知道能够在什么地方找到它。Emacs把这类目录都记录在全局变量`load-path`里，这个变量的值是一个由字符串元素组成的列表，每个字符串是一个目录名。

`load-path`的初始值只包含Emacs自带的LISP目录的名字，比如`“/usr/local/emacs/lisp”`。你需要把自己的LISP目录的名字添加到`load-path`变量里。这个操作可以用LISP函数`append`来完成，它的作用是把任意个数的列表参数合并到一起。假设你的LISP目录名是`“~yourname/lisp”`，那么就需要把下面这条语句添加到你的`“.emacs”`文件里：

```
(setq load-path (append load-path (list “~yourname/lisp”)))
```

函数`list`不可缺少，因为`append`函数的参数必须是列表。在`“.emacs”`文件里，这行代码必须出现在任何一条从LISP目录里加载程序包的命令之前。

当你试图加载一个开发库的时候，Emacs 会按目录在 **load-path** 变量里出现的先后顺序对它们进行查找；就我们刚才的设置情况而言，Emacs 会先查找它自己缺省的 LISP 目录。如果想让 Emacs 先查找 LISP 子目录，就应该用前面介绍的 **cons** 函数代替 **append** 函数，如下所示：

```
(setq load-path (cons "~/yourname/lisp" load-path))
```

当想用自己的程序包代替 Emacs 标准程序包时，这种格式很有用。例如，如果你自己编写了一个 C 模式版本，并想用它代替标准包，你就可以用这种形式。注意，这里的目录名不需要通过 **List** 函数调用，因为 **cons** 函数的第一个参数可以是一个原子项（在这种情况下是一个字符串），正如前面描述的计算模式，这种情况与使用 **cons** 函数将值压入堆栈类似。

如果想让 Emacs 查找你当前的工作目录，给 **load-path** 变量加上一个“**nil**”即可，可以用 **cons** 函数把它加在列表的开始，也可以用 **append** 函数把它加在列表的末尾。它的效果与把英文句号“.”添加到 UNIX 环境变量 **PATH** 里的情况差不多。

在创建了个人的 LISP 目录并告诉 Emacs 可以在什么地方找到它之后，就可以加载和使用自己开发的 LISP 程序包。有好几种方法可以用来把 LISP 程序包加载到 Emacs 里。下面给出了四种方法：我们在第十一章里曾经用过其中的第一种方法：

1. 输入用户级命令 “**ESC x load-library RETURN**”；参见第十一章。
2. 在 LISP 代码（通常就是你的 “**.emacs**” 文件）里加上语句 “**(load "package-name")**”。如果这条语句是加在你的 “**.emacs**” 文件里的，Emacs 就会在你启动它的时候加载指定的程序包。
3. 给用来启动 Emacs 的命令 (**emacs**) 加上一个命令行参数 “**-l package-name**”。这个操作将加载指定的程序包 *package-name*（注 10）。
4. 在 LISP 代码（通常就是你的 “**.emacs**” 文件）里加上语句 “**(autoload 'function "filename")**”，就像第十二章中介绍的那样。这个操作将在执行给定函数 *function* 时让 Emacs 自动加载相应的程序包。

注 10：另一个命令行选项 “**-f function-name**” 会让 Emacs 在启动时自动运行函数 *function-name*，此函数没有参数。

对 LISP 文件进行字节编译

在创建了自己的 LISP 子目录之后，如果你还对 LISP 文件进行了字节编译（byte-compiling），即把文件中的 LISP 转换成字节代码，就能更有效地加载和使用它们；字节代码是一种更加紧凑的机器阅读格式。对 LISP 文件 *filename.el* 进行字节编译将创建出一个字节代码文件 *filename.elc*。与没有进行字节编译处理的 LISP 文件相比，字节代码文件的长度通常会减少到原来的 40% 到 75%。

虽然经过字节编译的文件运行效率更高，但严格说来并不是一个必要的步骤。当 **load-library** 命令遇到参数 “*filename*” 时，它将先去查找是否存在一个名为 “*filename.elc*” 的文件。如果这个文件不存在，它就去查找 “*filename.el*” 文件（即未经字节编译的版本）是否存在。如果这个文件也不存在，它才会去查找 “*filename*” 文件。对 “*.emacs*” 文件也可以进行字节编译；如果你的 “*.emacs*” 文件比较大，字节编译之后的启动速度会有明显提高。

在编写 LISP 代码的过程中，可以对编辑缓冲区中的某个函数单独进行字节编译：把光标移到该函数定义里的任意位置，再输入 “**ESC x compile-defun**” 命令即可。如果想对整个 LISP 代码文件进行字节编译，请输入 “**ESC x byte-compile-file RETURN**” 命令和它的文件名。如果省略了 “*.el*” 后缀，Emacs 会自动加上它并请求确认。如果修改了文件但还没来得及存盘，Emacs 会先对它进行存盘。

当字节编译工作正在进行的时候，你将看到正被编译的函数的名字从辅助输入区里一闪而过。由字节编译器生成的文件其名字与原始的 LISP 文件相同，但后缀名中增加了一个字母 “*c*”；也就是说，“*filename.el*” 将变为 “*filename.elc*”，而 “*.emacs*” 将变为 “*.emacsc*”。

最后，如果你的 LISP 目录里有很多个 LISP 文件，而你只对其中的几个进行了修改，就可以用 **byte-recompile-directory** 命令只对那些从上次字节编译之后被修改过的文件重新进行编译（做法类似于 UNIX 操作系统的 **make** 工具）。你只要先输入 “**ESC x byte-recompile-directory RETURN**” 命令，再给出 LISP 目录的名字（或直接按下回车键以确认当前目录为缺省项）即可。



第十四章

Emacs 编辑器和 X 窗口系统

本章内容：

- Emacs 的 X 界面
- 让 Emacs 使用 X 字体和颜色
- 定制 Emacs 在 X 环境中的显示情况
- 通过 .Xdefaults 文件进行定制
- 属性、窗格、菜单和鼠标事件
- 与 X 服务器进行通信
- 良好的 X 程序设计风格

虽然 Emacs 能够运行在字符终端上，可它的本质却是 X 窗口系统下的一个客户软件，用不着借助 *xterm* 之类的终端仿真器就能运行。这就使它能够使用多种字体、能够解释鼠标事件、能够显示在多个 X 窗口里……总之，能够使用 X 窗口系统全套的功能。在这一章里，我们将介绍 Emacs 的 X 界面。

Emacs 的 X 界面

启动 Emacs 的时候，X 窗口系统的许多功能就已经处于随时待命的状态。我们先介绍一些马上就能使用的功能，然后再介绍一些关于 Emacs 定制和程序设计方面的内容。

窗口卷屏条和 X 状态行

运行在 X 窗口系统下的 Emacs 与运行在字符终端上的 Emacs 之间最明显的区别是每一个 Emacs 的 X 窗口都有它自己的 X 卷屏条。这些卷屏条与 *xterm* 窗口的卷屏条在行为上非常相似。可以用鼠标拖动卷屏块来连续卷动窗口内容，也可以在卷屏条里点击鼠标使窗口内容快速上卷或下卷。还可以点击或拖动窗口的状态行，其效果是对它上面或者下面的窗口进行移动以及调整窗口尺寸。

菜单条

Emacs窗口最顶部的那一行与Macintosh的菜单条很相似，上面依次排列着“**Buffers**（编辑缓冲区）”、“**Files**（文件）”、“**Tools**（工具）”、“**Edit**（编辑）”、“**Search**（查找）”和“**Help**（帮助）”等菜单标题。如果在这些标题上点击鼠标，就能打开各个菜单并通过鼠标轻松地选择执行各种常用的 Emacs 编辑命令。（从 Emacs 19.30 版开始，运行在字符终端上的 Emacs 也有菜单条了，但还是得通过键盘操作而不是鼠标来使用它们。）

比如说，“**Buffers**”菜单会给出一个 Emacs 编辑缓冲区的清单，用鼠标点击某个条目就可以进入相应的编辑缓冲区。“**Search**”菜单则把没有按键绑定的查找命令（标准的或使用了正则表达式的查找命令）都列了出来，可以通过鼠标来选择和执行各种查找操作。“**Edit**”菜单提供了 Macintosh 风格的剪切/粘贴操作。学习这些菜单的最佳方法就是多做练习。

如果不喜欢单击下拉菜单，想回到屏幕以前的显示方式，也可以禁用菜单条功能；具体做法我们将在后面讨论定制问题的章节里进行介绍。

注意：根据 Emacs 19.30（1996 年 6 月推出）的情况看，自由软件基金会好像还没有把 Emacs 菜单条的设计方案最终确定下来（这也是我们为什么没有在此介绍大量细节的原因）。因此，如果你的菜单条看上去与我们这里介绍的不一样，或者子菜单和菜单项发生了变化，请不必惊讶。

鼠标的其他用途

我们完全可以把鼠标点击动作绑定为任意的 Emacs LISP 函数，从而使鼠标的点击、拖放动作就像是输入了相应的键盘命令一样（我们将在本章后面的内容里对此做更详细的讨论）；不过，最常见的鼠标动作绑定情况还是帮助设置光标和文本块标记的位置。

如果在一个普通的 Emacs 编辑缓冲区里点击鼠标左键，Emacs 将会把插入点（即 Emacs 编辑缓冲区里的光标）移动到 X 光标位置。如果点击鼠标的地方没有文本（比如因为在该位置的左边已经有了一个换行符），Emacs 编辑缓冲区里的光标将被移动该行文本的最右位置（这类似于上、下方向键的动作效果）。

如果点击并拖动鼠标，就又是另外一种情况了。Emacs 会在发生点击动作的地方设置一个文本块标记，把插入点留在松开鼠标键的地方。也就是说，只需用一个点击-拖放命令就可以（为“**C-w**”等文本块编辑命令）定义一个文本块。这个文本块将立刻被复制到 X 窗口系统的选取缓冲区（selection buffer）和 Emacs 的删除环里。

当点击并拖动鼠标的时候，文本块标记和插入点之间的文本区域反显为灰色背景。当松开鼠标左键以设定插入点位置的时候，这个文本区域仍将呈灰色反显状态（注1）。通常，反显区域会在下一个输入事件发生时消失。可以让 Emacs 进入一个能够把反显区域保留在屏幕上（具体做法将在后面的内容里介绍）的编辑模式；但最好不要把它用做缺省的编辑模式，因为反显区域将随着插入点的移动而收缩或者扩大，从而给正常工作造成干扰。

鼠标中键缺省的绑定动作是插入 X 剪切缓冲区里的内容（与 *xterm* 窗口里的情况差不多）。

鼠标右键的作用是把文本块标记和插入点之间的文本复制到删除环里。如果在同一位置再次按下鼠标右键，文本块将会被删除。因此，用它来剪切和粘贴文本是很方便的。

按住 **ALT** 键再按下鼠标左键将对 X 窗口系统的辅助选取缓冲区（secondary selection buffer）进行操作，缺省的主选取缓冲区（primary selection buffer）不受影响。不过，知道辅助选取缓冲区的 X 应用程序并不很多，所以这个细节最好不用。

出于教学的目的，以上内容都稍微做了一些简化。它们没有提到双击或三击鼠标键时的各种特殊操作，也没有介绍各种特殊的文本块选取情况（这些细节可以通过 Emacs 的在线帮助系统查到）。在本书写作期间，自由软件基金会还在对鼠标接口进行微调；因此，鼠标的操作细节还有可能会发生变化，但我们在里介绍的操作行为应该不会再有变化了。

Emacs 的删除环和 X 的选取缓冲区

Emacs 内部的删除和恢复命令能够与 X 窗口系统的选取机制互通有无。用 “**C-w**”

注 1： 在 Emacs 19 某些早期的版本里，松开鼠标键的时候，灰色反显区域会消失。如果你遇到了这样的情况，就快去升级吧！

或“**C-k**”等命令删除 Emacs 文本时，被删除的文件也将被复制到 X 的选取缓冲区里；可以再通过标准的鼠标命令把它粘贴到其他 X 窗口里。

反过来说，X 选取的东西 Emacs 也看得见。对于遵守 X 窗口系统关于选取操作有关规定的应用程序来说，从它们的 X 窗口里剪切到的文本也将放入 Emacs 的删除环里，并且马上就能用“**C-y**”组合键或 Emacs 的其他删除恢复命令进行粘贴。

窗格的使用

在 Emacs 18 版本里，所有的 Emacs 窗口都只能显示在一个字符方式下的 X 窗口里。现在则不同了，Emacs 能够在一个或者多个窗格里进行显示；“窗格”是彼此分开的 X 窗口。

全体窗格都是从同一个给定的 Emacs 衍生出来的，驱动它们的也都是同一个基本进程；它们共享 Emacs 的二进制执行代码和 LISP 库的执行代码。因此，用一个 Emacs 打开多个窗格要比用多个 Emacs 分别打开它们更合算，前者加给机器的负载要比后者少。

Emacs 拥有一些对窗格进行操作的命令，我们把其中最常用的命令列在表 14-1 里。

表 14-1：Emacs 的窗格操作命令速查表

键盘操作	命令名称	动作
C-x 5 f	find-file-other-frame	访问另一窗格里的某个文件
C-x 5 d	dired-other-frame	在另一个窗格里运行 Dired 模式
C-x 5 0	delete-frame	删除当前窗格
C-x 5 b	switch-to-buffer-other-frame	切换到另一个窗格

正如大家看到的那样，这些命令与正常的文件和编辑缓冲区访问命令很相似（它们与文件访问命令的不同之处，只是多了一个不同的命令前缀）。

与窗格有关的其他函数和变量能帮助管理好 X 显示画面上的各种窗格，也能帮助 Emacs 的编辑模式用好窗格。如果你对这些函数有兴趣（它们都没有缺省的按键绑定），可以用“**ESC x apropos frame**”命令来查阅；它们每一个都有在线帮助信息。

让 Emacs 使用 X 字体和颜色

X 窗口系统下的 Emacs 能够以多种固定宽度的字体来显示文本。它对比例间隔字体的处理目前还做得不是很好，希望以后的版本能够解决这一问题。X 窗口系统下的 Emacs 还能在 X 服务器的支持下以多种前景色和背景色的组合来显示文本。

按住 **SHIFT** 键再点击鼠标左键将打开一个字体菜单。在做出选择的同时，当前窗格里的 Emacs 字体会立即发生改变并重新刷新这个窗格。可以用这个方法来检查不同字体在屏幕上的显示情况，看哪几种字体的可读性最好。

如果想查看可用颜色的范围，请从菜单条上选择“**Edit**”菜单，打开“**Text Properties**（文本属性）”子菜单，再选择“**Display Colors**（显示颜色）”项。虽然这个菜单项不会把所有可能的前景色和背景色组合都显示出来（它们实在是太多了，没有几千，也有几百），但已经足够为文本挑选前景色和背景色之用。

自动反显与着色功能

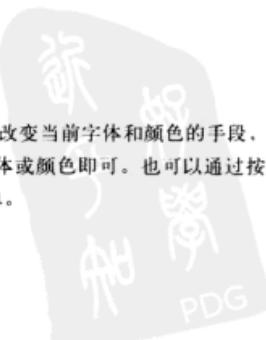
要想使用字体和颜色，最简单的方法就是加载 LISP 程序包 **font-lock.el**（这是 Emacs 发行版本自带的一个程序包）。这个编辑模式可以用不同的颜色和字体把文本编辑缓冲区里的不同部分标识出来。比如说，可以把 C 和 LISP 编辑缓冲区里的程序注释都挑出来，给它们加上颜色，让它们与黑色文字的程序代码有所区别。如下所示：

```
;; Turn on font lock mode every time Emacs initializes a buffer
;; for Lisp or C.
;;
(add-hook 'emacs-lisp-mode-hook 'turn-on-font-lock)
(add-hook 'c-mode-hook 'turn-on-font-lock)
```

font-lock 模式特别适用于给程序代码或大纲模式文本着色，对其他高度结构化的文本也非常有用，比如 **troff** 稿件的源代码等。

改变文本的字体和颜色

Emacs 菜单里的“**Edit**”下拉菜单提供一种方便地改变当前字体和颜色的手段，只需从“**Text Properties**”子菜单里选一个新的字体或颜色即可。也可以通过按住 **CTRL** 键再按下鼠标第二键的方法来打开这个菜单。



为了用好“**Text Properties**”菜单，需要知道Emacs在其内部是用术语“字型”(face)来描述字体和颜色的。一种字型就是字体和颜色的一种组合。“**Text Properties**”菜单准备了一些事先安排好的字型和一个用来通过名字指定其他字型的选项。

我们将在本章后面的内容里对字型、给字型起名字以及有关的LISP程序设计结构做进一步的讨论。就目前来说，可以简单地认为编辑缓冲区里的每个字符都有一个不同的字型以不可见的方式与它关联即可（当然，在实际工作中，字型很少会变化得如此频繁）。

保存设置了字体和颜色的文本

细心的读者可能已经注意到了，虽然着色机制允许我们给编辑缓冲区里的文本加上颜色，可我们却没有介绍怎样才能把文本属性保存起来供今后的编辑工作使用。这是一个很重大的问题。如果没有办法把属性和文本一块儿保存起来，那么再好的字体和着色机制也都不过是画面显示方面的一种噱头而已，用来装饰编辑缓冲区倒是不错，可在加强 Emacs 编辑能力方面却并没有多大功效。

我们需要这样一种方法来解决这个问题：把文本属性当做一种文本排版标记 (text-markup) 保存起来；然后，当下次编辑文件的时候，再把文本排版标记恢复为文本的属性。

在本书的写作期间，这方面的试验性代码已经添加到Emacs里。一个名为**enriched-mode**的程序包允许人们把文本属性保存为一种符合因特网技术标准RFC 1563文档有关规定的MIME扩展文本格式（注2），然后再把这种格式的文件按照原来的文本和文本属性导入 Emacs 编辑缓冲区。这个编辑模式听起来很不错，但要想让它达到实用要求并做到能够与 Emacs 的其他编辑模式（比如邮件处理器和 WWW 浏览器等）完善地集成在一起，在其设计和开发方面还有很多工作有待完成。等大家读到这本书的时候，这样的程序包可能已经有好几个了，每个程序包支持一种不同的扩展格式，比如HTML格式等。这类编辑模式的发展方向是让 Emacs 具备支持WYSIWYG 编辑能力，甚至具备支持多媒体编辑的能力。

注 2： RFC 1563 已经被 RFC 1896 取代。Emacs 使用的标记方法看来应该是遵守 RFC 1896 的。

定制 Emacs 在 X 环境中的显示情况

让 Emacs 给编辑缓冲区里的文本进行着色的方法我们已经介绍过了。X 环境下的 Emacs 还有其他一些值得一用的定制方法。如果想把屏幕上原来是菜单条的那一行省下来以增加一个文本行，可以把下面这条语句添加到 “*.emacs*” 文件里：

```
(menu-bar-mode nil)
```

当然，也可以通过辅助输入区完成这一定制工作。

还可以让 Emacs 为每个编辑缓冲区分别创建一个它自己的窗格。为了实现这个定制，需要把下面这条语句添加到 “*.emacs*” 文件里：

```
(setq pop-up-frames t)
```

是否需要做这样的定制完全取决于个人。在 X 桌面空间的使用习惯方面，我们基本上可以划分出（至少）两种不同的风格。第一种风格的用户总是在相对固定的位置打开几个大的窗口：一个窗口里可能是个编辑器、另一个窗口里可能是个时钟等等。采用这种桌面布局的人通常喜欢让桌面保持整洁。如果你把自己归入这类人群中，把这个变量设置为 “t”的效果大概不会让你喜欢。而第二种风格的用户似乎更喜欢桌面堆满文件。如果你就是一名这样的用户，那么 **pop-up-frames** 就是为你准备的。

函数调用

```
(transient-mark-mode t)
```

将改变 Emacs 在选取文本块时的反显行为，使灰色反显背景总是出现在插入点和文本块标记之间的区域里。在这种模式下，只要再开始对编辑缓冲区做修改，它的文本块标记就将被取消；这是为了让屏幕上不至于总是有一大块灰色的区域干扰用户的正常工作，特别是当我们知道文本块的边界已经不再有用的时候。

通过 *.Xdefaults* 文件进行定制

还可以用在 *.Xdefaults* 文件（或者是 *.Xresources* 文件）里设置 X 资源的方法对 Emacs 进行定制。我们准备介绍几个比较重要的 X 资源；如果想查看允许进行设置的资源的清单，请参考 GNU Emacs 的使用手册页。

要想对 X 资源进行设置，必须编辑资源文件。需要为想设置的每一项资源添加一条下面这样的语句：

```
emacs.<resource>:<value>
```

如果想对 geometry（几何）资源（即窗口的尺寸和坐标）进行设置，就必须使用下面这样的语句：

```
emacs.geometry := 80x49 + 270 + 0
```

这条语句的作用是把 geometry 资源设置为值“=80x49+270+0”。它告诉 Emacs 使用一个 80 个字符宽、49 个字符高的窗口，窗口从屏幕点阵坐标 (270, 0) 处开始显示。在一个分辨率为 1024x768 的显示器上，如果使用 14 磅字体并且窗口不带边框，这样的设置将使 Emacs 窗口占据整个显示器右面几乎四分之三的面积，垂直方向上的文本行数则是尽可能的多。可以根据自己的具体情况来修改这个例子里的数值。几何资源的详细计算方法请参考 X 的使用手册页。

font 资源允许选择一种显示字体供 X 环境下的 Emacs 使用。注意只有固定宽度的字体才能正常工作。请看下面的例子：

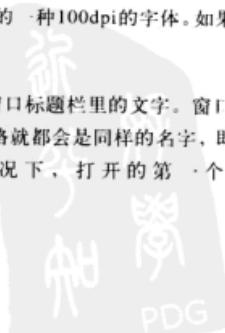
```
emacs.Font: *fixed-bold-r--14--*
```

这个例子设置的显示字体是 14 磅、每英寸 75 点（即 75dpi），这种字体在 17 英寸、0.31 点阵、1024x768 分辨率的显示器上工作得很好。如果点阵数更小（比如 0.28 点阵）或者显示器更大，就可以设置一个更小的磅值，Emacs 窗口将能容纳更多的文本行。如果你有一台高级的 21 英寸、0.25 点阵的显示器，就可以试试下面这样的设置：

```
emacs.Font: -*courier-*r-*--16-*100-100-*-*-*--*
```

这是专为点阵数非常小的显示器而准备的一种 100dpi 的字体。如果你觉得小字体读起来费眼，就挑选一种大字体好了。

通过 title 资源可以设置显示在 Emacs 窗口标题栏里的文字。窗口标题一般不用设置；如果进行了设置，所有的 Emacs 窗格就都会是同样的名字，即设置的标题；如果没有设置，那么，在缺省的情况下，打开的第一个窗格的名字是



“`emacs@hostname`”（“`hostname`”将被替换为你机器的主机名），以后创建的窗格将使用其中的编辑缓冲区的名字。

foreground（前景）和 **background**（背景）资源允许对 Emacs 窗口里的文本颜色和背景颜色进行设置。资源值必须是标准的 X 颜色的名称。如果临时标记的文本块反显功能看着不对劲，就可能需要对这两项资源进行设置。

修改 `.Xdefaults` 文件之后，需要在主目录里运行下面这条命令：

```
xrdb .Xdefaults
```

或者在任何一个目录里运行下面这条命令：

```
xrdb ~/._Xdefaults
```

如果你觉得采用命令行参数来设置这些资源的方法更方便，也可以那样做。比如说，**-font** 选项所使用的字体设置参数与我们刚才介绍的 `font` 资源所使用的完全一样（但必须给它加上引号以免 shell 对其中的元字符进行扩展）。可以用 **-geometry** 选项设置几何资源，用 **-fg** 和 **-bg** 选项设置前景色和背景色。其他可用的选项请参考 Emacs 的使用手册页。

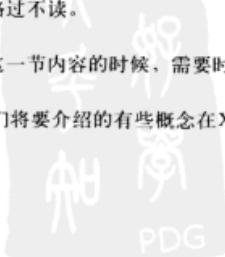
属性、窗格、菜单和鼠标事件

下面，我们将展开关于使用 Emacs LISP 进行 X 程序设计方面的讨论，这部分内容与第十三章是相互衔接的。通过前面对 X 定制方面的介绍，大家可能已经对 Emacs 的 X 接口代码的工作情况有了一个直观的了解。为了把 X 环境下的 Emacs LISP 程序设计讲明白，我们对这些概念的探讨还需要再稍微正式一些、细致一些。

在这一节里，我们将对几个新出现的与 Emacs 的 X 支持功能有关的基本概念进行讨论。如果你对 Emacs LISP 程序设计没有兴趣，则本章从这里开始一直到结尾的内容都可以略过不读。

在阅读这一节内容的时候，需要时刻记住下面这 3 件事情：

- 我们将要介绍的有些概念在 X 或 GUI 环境以外的地方也有意义（特别是菜单和



文本属性)。但在我们写作本书的期间,字符终端上的 Emacs 还只能零星地提供一点对它们的支持——如果真的能支持它们(注 3)。

- 每个新概念通常都会引入一个新的 Emacs LISP 数据结构。弄清每种数据结构都有哪些操作具有实际使用价值,这对加深我们理解其行为是很有帮助的,弄清哪些操作开销小,哪些程序开销大也很有必要。
- 自从 Emacs 19 的首次发行以来,某些与 X 有关的函数的行为和某些与 X 有关的数据结构的形状已经发生了变化(出现了一些不互相兼容的情况)。虽然未来版本发生巨大变化的情况就目前看不太容易发生,但多加注意和勘查《Emacs LISP Reference Manual》(Emacs LISP 参考手册,自由软件基金会出版)或 Emacs 的在线文档是很值得推荐的做法,如果你是在这本书最后一次再版好几年之后才读到它的,就更要注意这一点。

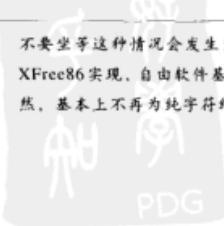
窗格

X 环境下的 Emacs 最简单的基本构造是“窗格”。如果你在阅读此前各章节的时候动了脑筋,对窗格及其功能就应该已经有了一个大致的了解。X 环境下的 Emacs 有一个缺省的窗格。一个窗格就是一个显示一个或者多个 Emacs 窗口的 X 窗口。全体窗格共享同一个编辑缓冲区清单和其他 Emacs 上下文。

窗格是用一种基本数据类型实现的,这种数据类型的名字就叫做窗格。用来创建新窗格的函数只有一个, **make-frame**。它的输入参数是一个给定格式(具体格式可以在《Emacs LISP Reference Manual》里查到)的关联列表,它对窗格的大小、位置、X 字体以及各种其他参数做了定义。缺省值取自一个 Emacs LISP 全局变量 **default-frame-alist** 和 X 的资源系统。与窗格有关的其他函数通常会以这些窗格对象为参数(而且往往是第一个参数)。

函数 **redraw-frame** 的作用是绘制窗格,而 **delete-frame** 函数的作用是删除窗格。如果想查看所有窗格,可以检查 **frame-list** 和 **visible-frame-list** 函数的返回值(但不能用修改这些列表返回值的方法来创建或者删除窗格,它们只是 Emacs 内部窗格列

注 3: 不要坐等这种情况会发生变化,因为已经出现了适用于 Linux 和其他免费 UNIX 的 XFree86 实现,自由软件基金会和其他一些 Emacs 程序员现在都把 X 支持看做理所当然,基本上不再为纯字符终端做开发了。



表的副本)。为了掌握实际窗格列表 (Emacs Lisp 的内部列表, **frame-list** 是它的一个副本) 中的窗格次序, 可以使用 **next-frame** 和 **previous-frame**。

在任一给定的时刻, 只能有一个窗格被选中。只有被选中的窗格才能接受输入事件。各种窗格函数通常用一个取值为 “nil”的窗格对象参数来代表当前的选中窗格。可以用窗口管理器为 X 窗口系统提供的任意一种方法选中一个新的窗格, Emacs 会知道发生了什么变化, 还可以通过 **select-frame** 函数以程序控制的手段强行切换窗格, 强制 X 把输入焦点交给新窗格。

窗格可以处于可见状态、不可见状态或图标化状态 (被缩小为一个 X 图标, 点击图标将使它还原为窗格)。可以通过窗口管理器以交互方式让窗格进入这三种状态。Emacs LISP 代码可以用函数 **make-frame-visible**、**make-frame-invisible** 和 **iconify-frame** 来改变窗格的显示状态。窗格的显示状态可以用 **frame-visible-p** 函数测试出来。

窗格的大小可以用 **frame-height**、**frame-width**、**frame-pixel-height** 和 **frame-pixel-width** 等函数分别查到; **set-frame-size** 函数可以设置窗格的大小。各种其他参数可以通过 **frame-parameters** 函数所返回的关联列表查出来; 其格式与 **make-frame** 函数的参数格式完全一样。可以通过 **modify-frame-parameters** 函数改变这些参数中的任何一个。各种窗口参数的含义和作用在《Emacs LISP Reference Manual》(Emacs LISP 参考手册) 里有详细的说明。

窗格的程序设计其实挺简单 (但所涉及的鼠标和焦点事件方面的理论却相当复杂, 大家学到后面就能亲眼见到)。最重要的事情是必须记住有哪些 Emacs 全局变量是适用于全体窗格的, 又有哪些是各窗格专用的局部参数。一般说来, 只要是能够在屏幕上看到的任何东西、或者说与 X 有关的任何东西, 各个窗格就都会分别有一个彼此互不影响的值, 这些值是创建窗格时从系统的某个缺省设置值那里复制过来的。

文本属性与覆盖

Emacs 19 表示文本属性 (text properties) 的基本表示法可以附着在编辑缓冲区或者字符串里的字符上。当字符被移动或者被复制时, 它的文本属性也会随之转移。编辑缓冲区里的文本的属性被认为是编辑缓冲区内容的组成部分; 对它们进行的修改也可以像被改动的文本那样通过撤销操作恢复回来。

Emacs 还允许你定义自己的文本属性，这对某些场合来说是非常有用的。在使用文本属性方面的功能时，很多用法都涉及到文本属性的一个子集，这个子集里的文本属性对 Emacs 的显示管理器（display manager）和编辑命令有特殊的含义。我们将在本章后面的内容里对这些属性做进一步的讨论。

在 Emacs 的内部，字符属性是用一个由 3 个元素构成的列表来表示的，这 3 个元素分别是：一个把文本属性的名称和值关联起来的列表，一个给出文本属性作用范围起始点的编辑缓冲区索引或字符串索引，和一个给出文本属性作用范围结束点的编辑缓冲区索引或字符串索引。之所以采用这种表示形式的列表（而不是把每个字符位置上的文本属性分别保存起来），部分原因是为了避免用来保存文本属性的列表体积过大，或者说是为了避免让指向该列表的假名指针过多，同时还能做到把查找某种属性的平均开销控制在可接受的范围内；但更主要的原因是为了能够迅速查找到文本属性的值发生了修改的地方。

与文本属性有关的基本函数

为了对文本属性进行测试和设置，Emacs 准备了一套完备的函数。函数 **get-text-property** 允许查询某已知属性在给定位置处的值；而 **set-text-property** 函数则允许设置一项属性；函数 **all-text-properties** 允许查看由给定位置处的全体属性组成的一个列表。在程序设计实践中，需要对编辑缓冲区中的某个区域或者某些字符串位置的属性值进行检查和设置的情况往往更多见一些；这些工作可以用 **add-text-properties**、**put-text-property**、**remove-text-properties** 和 **set-text-properties** 等几个函数来完成。关于这几个函数的详细资料请查阅《Emacs LISP Reference Manual》。

有时候，我们需要找出编辑缓冲区或字符串某个给定的属性值是从什么地方开始发生变化的。事实上，因为这个操作对 Emacs 显示管理器来说的确是太重要了，所以人们对根据编辑缓冲区的内容而提取其文本属性列表的操作进行了彻底的优化。如果你想看看这些优化的效果，可以用函数 **get-text-property** 和其他有关函数，特别是 **next-property-change** 和 **previous-property-change** 函数，或者 **next-single-property-change** 和 **perevious-single-property-change** 函数对编辑缓冲区进行遍历。这些函数的细节也可以从《Emacs LISP Reference Manual》中查到。



特殊属性

最重要同时也是最常用的文本属性是字符的“字型 (face)”。不带字型属性的字符将被显示为当前 Emacs 窗格所使用的缺省 X 字体 (可以随时用“**C-mouse-2-down**”命令改变这个 X 字体)。通过对字符的字型属性进行设置，就可以改变它的 X 字体、颜色或者有没有下划线等属性。我们已经介绍过从“Edit”菜单对这些属性进行设置的方法；还可以利用 Emacs LISP 提供的一些函数在程序控制下实现同样的设置功能。

local-keymap 属性允许把一个局部键位映射图和一个字符序列关联起来。这个属性最适合应用于某种文本属性在编辑缓冲区内的作用范围的起始点索引或结束点索引，这样，就可以创建一些在编辑缓冲区的不同区域里有不同绑定的特殊热键。

属性 **read-only** 可以禁止对某个字符或某个区域进行写操作；如果有人想对它进行修改，就会引发一个 LISP 错误。

特殊属性 **modification-hooks**、**point-entered** 和 **point-left** 允许对挂钩函数做这样的设置：每当某字符范围被修改或者被插入时，就执行这些挂钩函数。此外，为了能够以间接方式对文本属性进行设定，Emacs 还专门准备了一个 **category** 挂钩。这些特殊属性的细节超出了我们现在对 Emacs 的 X 界面的讨论范围；如果读者需要了解这方面的详细资料，请自行阅读《Emacs LISP Reference Manual》(Emacs LISP 参考手册)。

字型的设置和命名

创建新字型对象的基本函数是 **make-face**。创建新字型之后，还需要对它的属性进行设置，其中最重要的一个属性是字体。给字型设置字体的工作可以用函数 **set-face-font** 来完成，这个函数的输入参数是一个字符串，而这个字符串必须是一个合法的 X 字体设置项 (即该字符串的语法必须与 X 资源文件所使用的有关语法一致)。

我们还能对字型某些其他特征进行定制，其中包括字型的前景色、背景色、前景点阵、背景点阵以及该字型是否带有下划线等。这些属性中的每一个都有一个与之对应的 **set-face** 函数，而这些函数的详细资料都可以通过 Emacs 的在线求助系统查到。

在引用某个字型对象的时候，既可以使用一个字符串形式的名字 (即给 **make-face**



函数准备的参数),也可以直接使用该字型对象的一个副本;这与编辑缓冲区的情况有些类似。我们推荐使用第一种方法(用名字来引用字型对象)。Emacs的有关文档没有对“给某个类型为字型的值设置的属性是否会影响到与之同名的其他字型对象”这一问题做出明确的说明。

注意: 在我们写作本书的时候(1996年6月),这里探讨的某些细节还没有收录到当时的《Emacs LISP Reference Manual》里。请大家留意可能的修订,千万不要麻痹大意!

覆盖

对编辑缓冲区里的某些文本区域而言,它们的一些字型属性不会被看做编辑缓冲区内容的一部分(因此,移动和复制操作也不会保留它们的字型属性),我们把这些字型属性称为“覆盖(overlay)”。比如,在某些主要用途是编辑表格或者编辑列表清单的主要编辑模式里,为了以多种反显效果和多种颜色绘制出这些编辑模式的显示画面,就可能需要用到覆盖。

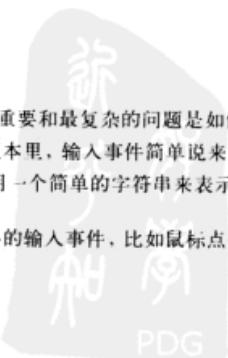
一个编辑缓冲区可能有多种覆盖;每种覆盖都有一个优先级,而高优先级的覆盖将取代低优先级的覆盖。版式的优先级高于文本属性。你可以设定某种覆盖只在某个特定的窗口里起作用,而不是访问同一编辑缓冲区的所有窗口。

函数 **make-overlay** 和 **delete-overlay** 的作用分别是创建和删除覆盖。查看和设置覆盖属性的函数分别是 **get-overlay** 和 **put-overlay**。函数 **overlay-start**, **overlay-end** 和 **overlay-buffer** 允许对覆盖的作用范围进行检查; **move-overlay** 则用来改变其作用范围。最后, **overlays-at** 和 **next-overlay-change** 可以帮助查找到指定的覆盖和它们的边界。

X 窗口系统中的输入事件

在 X 环境下, Emacs LISP 必须面对的最重要和最复杂的问题是如何表达和处理输入事件。在基于字符终端的 Emacs 早期版本里, 输入事件简单说来就是字符,只要是 Emacs 收集到的输入事件,就都可以用一个简单的字符串来表示。

但 X 环境下的 Emacs 却必须对付种类更多的输入事件,比如鼠标点击动作、鼠标移



动动作、切换窗格焦点等等。相应地，输入事件的内部表达方式也就越来越复杂，适用面也越来越宽。除了能够把函数绑定为普通的组合键，或者组合键加修饰键之外，现在还可以把函数绑定到用来表示 X 输入事件的向量上，而每一种输入事件可能都有一种它自己的结构。以前只接受按键组合为参数的函数（比如 **define-key**、**global-set-key** 和 **local-set-key** 等）现在也开始接受这些输入事件向量了。

输入事件主要分为 4 大类：按键组合、功能键、鼠标事件和窗格焦点事件。我们将在下面依次对它们进行讨论。

按键组合

输入数据中的 ASCII 字符是用 ASCII 字符来表示的。但字符编码的宽度却是 16 位（因而允许使用诸如 UNICODE 之类的大字符集），而且还可能会额外带有几个修饰位，这些修饰位包括 **META**、**CTRL**、**SHIFT**、**HYPYER**、**SUPER** 和 **ALT**。

有几个修饰位的名称似乎暗示着它们携带的信息已经被编码到输入事件的 ASCII 部分里，而你也许正在奇怪 **META** 和 **ALT** 到底有哪些修饰作用；如果真是这样，就说明你被一些聪明但却完全错误的推论误导了——但这并不是你的过错。为了把这个问题解释清楚，我们需要回溯一下“古老的”历史。

很久以前，当 Richard Stallman 还在麻省理工学院的人工智能实验室里工作的时候，一种被人们称为 LISP 机器的分析引擎出现了。用 LISP 语言实现的 Emacs 早期版本就运行在这些机器上，它们对此后（包括 GNU Emacs 在内）的 Emacs 版本的设计有着巨大的影响。这些计算机的基本数据类型是 32 位的标记字 (tagged word)，其中 8 位给出类型信息或其他特殊信息，另外 24 位给出数据或地址。LISP 机器上的每一个按键都会在输入流 (input stream) 里被转换为一个 32 位的数据项，数据项的扩展位表示它们是字符。很明显，ASCII 字符只能填满 24 位符号空间的一个小角落。

后来，人们又为这些 LISP 机器发明了一种专用键盘。除常见的 **CONTROL** 和 **SHIFT** 修饰键以外，这种键盘还另外多出 5 个修饰键 (**META**、**HYPYER**、**SUPER**、**TOP** 和 **FRONT**)。这些修饰键中的每一个都对应着基本 ASCII 按键所产生的代码中的一个或者多个修饰位。

专用键盘上的许多按键都在键帽上画有3个符号：顶上是一个字母和一个符号，前面是一个希腊字母。比如说，“L”键的顶上画着一个字母“L”和一个左右方向的箭头符号，前面是希腊字母“λ”。如果在左手按住某个修饰键的同时再用右手按下这个键，就可以输入箭头符号或拉丁字母“L”与希腊字母“λ”的大、小写形式。**CTRL**、**META**、**HYPERR** 和 **SUPER** 键的任意组合都能输入这几样东西。

事实上，这种专用键盘可以产生八千个不同的字符；这就使它们能够轻松地输入复杂的数学公式或者代表上千种单字符命令。为了节省输入时间，很多程序员都愿意多记住一些按键的命令含义（Emacs的操作界面就在这种氛围里形成了）。

修饰键实际上控制着机器字数据部分高8位（第17到第23位，统称为“修饰位”的翻转状态。以“a”键为例，是否需要把带有“control”修饰位的“a”键解释为ASCII字符“CTRL-A”完全取决于操作系统。

到了20世纪80年代后期，LISP逐渐退出了历史舞台；通用机器已经能够以更快的速度运行LISP。但对LISP机器专用键盘的记忆却延续了下来，X窗口系统的**keysym**程序在设计思路上就受到了它的影响，可以利用这个工具把任意一个键定义为修饰键，也可以利用它来定义各种古怪的组合键。当Richard Stallman准备重新布置用于X环境的Emacs键盘命令时，LISP机器专用键盘的影子又出现了。

这就是把输入事件代码的第18到第23位分别设定为**META**、**HYPERR**、**SUPER**、**SHIFT** 和 **ALT** 按键修饰位，并且仅在输入事件无法用ASCII字符表示的时候才使用的原因。比如说，代表ASCII大写字母“A”的输入事件就不需要对**Shift**修饰位进行置位，代表“C-a”组合键的事件也不需要对**CTRL**修饰位进行置位；可如果按下的是“C-M-A”（即**control-meta-shift-a**）组合键，则因为ASCII无法对它进行表示，所以**META**、**CTRL** 和 **SHIFT** 修饰位就都将被置位。

这段历史也道出了Emacs文档混用“M-”和“ESC”的原因（为了避免出现这种混乱，本书统一使用“ESC”来描述组合键；这与自由软件基金会文档里的做法正好相反，那些文档里到处都是“M-”）。用来设计Emacs的机器原本有一个**META**键，整个操作接口就是围绕着它而设计的。当Richard Stallman和朋友们把Emacs移植到没有**META**键的机器上去的时候，他们采用了让“ESC”前缀等同于一个**META**换档修饰位的简单做法，但保留了用来描述按键组合的老记号办法。

各修饰键的完整定义可以在Emacs的在线文档里查到。随着时间的推移和程序设计技术的发展，LISP专用键盘上需要使用4个修饰键的命令变得不那么流行了；现在的程序员可能永远不用了解这些修饰键的细节了。

功能键

功能键在按下时将被表示为一个符号而不是一个字符，而这个符号就是该功能键的名称。大多数名称符号的含义都是很明显的（比如4个方向键分别代表着上、下、左、右四个方向，“f1”代表着第一个功能键等等）。

字符终端下的Emacs 19就采用了这种做法。Emacs会根据终端的termcap设置项对功能键的定义情况做出分析，再把它们（虽然你看不见）添加到自己的输入分析表里；这样，那些功能键就能返回正确的符号。但在Emacs 18里，与终端有关的LISP程序包都是自己提供各种终端转义序列，这些程序包现在几乎都被淘汰了。

如果需要表示功能键和某个修饰键的组合，可以给功能键的符号加上相应的前缀，比如**META**用“M-”来表示，**SHIFT**用“S-”来表示，**ALT**用“A-”来表示。**CTRL**用“C-”来表示等等。这些前缀的顺序无关紧要，比如说，“C-S-f1”与“S-C-f1”的效果完全相同。

鼠标事件

鼠标事件是用列表表示的。列表的头元素是一个鼠标按键符号，即“mouse-1”、“mouse-2”或“mouse-3”（在用**define-key**或其他有关函数设置鼠标绑定时，只需给出这3个符号就足够了）。这些鼠标符号也可以像键盘按键那样加上修饰位前缀。

用来表示鼠标事件的列表里可能还包含一个或者多个地址元素。地址元素也是一个列表，它负责给出以下细节：发生鼠标事件的窗口和编辑缓冲区，事件发生时的鼠标位置，本次事件与上次事件之间的时间间隔（以微妙为单位）等。

最简单的鼠标事件是在不移动鼠标的情况下按下并释放鼠标——即“点击事件”，列表头元素符号的后面将只有一个地址元素。这个事件将在你松开鼠标按键的时候被发送出来。

如果在按住鼠标键的同时还拖动了鼠标，就会在松开鼠标时得到一个“拖动事件”。拖动事件有两个地址元素（分别对应于事件的起始点和结束点），并且会在头元素符号的修饰位前缀后面再加上一个“**drag-**”前缀。不过，如果没有把拖动事件绑定为某种操作，Emacs就会丢弃第二个地址元素并把它表示为一个点击事件。这就是说，不用你来区分短距离的拖动事件与点击事件，除非你真的想这么做。

还有一个环节。按下一个鼠标键的时候，将得到一个“按下事件”。它与点击事件很相似，但事件列表头元素符号在鼠标键符号，和它的修饰位前缀之间会多出一个“**down-**”前缀。鼠标事件通常都是配对出现的，所以按下事件的后面要么会跟有一个点击事件，要么（比较少见）会跟有一个拖动事件。如果需要对这些细节进行处理，可以给函数“(**track-mouse ...**)”包上一段 LISP 代码做为“外套”，用这个“外套”来处理鼠标移动事件；表示鼠标移动事件的列表有一个头元素符号“**mouse-movement**”和一个地址元素。

最后，我们再说说关于键位映射图的事。以鼠标事件开始的按键组合将根据发生鼠标动作的编辑缓冲区的键位映射图来做出解释，而这个编辑缓冲区并不一定是当前选中的编辑缓冲区。这条规定的效果是很直观的：可以点击另外一个编辑缓冲区里的文本并执行一些特殊的命令，而这些操作将在不丢失或者不需要改变编辑缓冲区焦点的情况下得以完成（除非鼠标绑定被显性地设置为改变编辑缓冲区的焦点——而这是比较常见的做法）。

为了把这些概念解释得既明白又简单，我们在这里省略了许多具体的细节。这些细节大多与 **read-key-sequence** 函数针对输入事件的解释方法，或者说与 Emacs 自身的向后兼容性有关，这是为了保证 Emacs LISP 代码中那些包含“**\M-**”修饰位的按键组合字符串仍然能够使用。如果需要了解这方面的知识，请查阅《Emacs LISP Reference Manual》（Emacs LISP 参考手册）。

窗格焦点事件

窗格焦点事件并不复杂。当用户把输入焦点从一个 Emacs 窗格，切换到另外一个窗格里去的时候，就会产生一个窗格焦点事件，用来表示窗格焦点事件的列表由两个元素组成，一个是头元素符号“**switch-frame**”，另一个是新窗格的名字。

当前输入事件只有在整个序列完整地输入之后才会被报告出来。这种延迟是为了避

免出现某些混乱。比如像用户开始输入一个复杂的序列却在半中间不小心改变了当前窗格的场合。按键组合将按“老”窗格的键位映射图进行解释，除非事先发送过焦点改变事件。

菜单

菜单是一种特殊结构的键位映射图，这种键位映射图允许为有关操作提供提示字符串。换句话说，它不仅能够把一个函数或者另外一个键位映射图绑定到一个按键组合上，还可以把一个函数和一个提示字符串同时绑定到一个按键组合上。当菜单显示时，那个提示字符串将成为菜单项的名字。每个菜单键位映射图还可以有一个总的提示字符串，它将用做该菜单的标题。

要想激活一个菜单，就必须把它绑定到一个起操作前缀作用的组合键上。菜单键位映射图在工作原理方面与普通的键位映射图没什么两样，但在动作行为上却有所不同：跟在操作前缀后面的各输入事件将弹出相应的菜单（或子菜单）。既可以用传统方法（键盘操作）选择菜单项，也可以用鼠标点选。

对菜单功能的开发目前仍在进行当中。除了用做菜单项名字的提示字符串之外，还可以给每个菜单项关联上一个帮助字符串；也就是说，在线求助功能可以用来给每一个菜单项提供相应的帮助信息。

Emacs 不仅准备了几种用来布置菜单格局的方法，还准备了一种能够把你的菜单添加到 Emacs 窗格顶部的菜单条上的方法。如果你对这些方法感兴趣，或者想了解键位映射图和菜单的 LISP 表示法方面的细节，请自行查阅《Emacs LISP Reference Manual》（Emacs LISP 参考手册）或 Emacs 发行版本自带的 /etc/LNEWS 文件。

与 X 服务器进行通信

Emacs LISP 还提供了很多用来与 X 服务器进行通信的函数。想用这么短的篇幅把它们都介绍给大家是不现实的，所以我们只拣其中最重要的几个说说，好让大家对它们有一个初步的了解。

要想查出关于 X 服务器显示能力方面的信息，可以使用的函数有 **x-display-pixel-**

`height`、`x-display-pixel-width`、`x-display-mm-height`、`x-display-mm-width`、`x-display-planes`、`x-display-backing-store`、`x-display-color-cells` 和 `x-display-color-p` 等等。还可以用 `x-rebind-key` 函数修改键盘按键所发送的字符序列。

一般说来，与 X 环境有关的函数在名字里都有一个“`x-`”前缀。所以，如果把“`x-`”提供给 Emacs 帮助系统的 `apropos` 函数做为其参数的前缀，就能查出与 X 环境有关的函数的完整清单。

良好的 X 程序设计风格

在前面介绍窗格对象的小节里，我们提到过这样一件事：有几个与窗格有关的 Emacs LISP 函数完成的其实是一些通常由 X 窗口管理器负责的操作，比如窗口的图标化。如果你去学习《Emacs LISP Reference Manual》(Emacs LISP 参考手册)，就会发现还有其他一些这样的函数我们没有在这里讲过，例如，从 Emacs LISP 直接读、写 X 服务器的选取缓冲区是完全能够做到的。出于其自身的设计理念，Emacs LISP 提供一些这样的函数，但如果你真的想把这些函数用在一个本身并不是由 X 输入事件循环驱动的 Emacs LISP 程序里，就一定要三思而后行。传统交互式程序（包括运行在字符终端上的 Emacs 在内）的 I/O 风格都是程序化的：它们被设计为每次只使用一个，而且只有在它们工作正常并准备接受输入数据时才明确地请求用户进行输入。而程序则会根据用户在整个执行过程中的具体位置划分为许多个状态。这类程序的交互式执行过程，可以看做由许多依次进行的 I/O 小循环构成的集合体，每个 I/O 小循环面对的是全部输入事件的一个子集，而当前的输入事件子集应该与前一个有所区别。

但 X 世界里的情况就不一样了，几乎所有的程序（还都是些“好”程序）都是由一个巨大的随机事件循环驱动的，这个循环能够接受的 X 输入事件范围相当广泛，不仅包括普通的按键组合，还包括鼠标点击、窗口的打开与关闭事件、窗口尺寸调整等许多内容。只有很少的程序状态被编码为全局性的执行序列；与程序运行状态有关的变量就更多了，与这些变量有关的内部检查也更多了。如果用户准备采取某项行动，可它的某些先决条件还没有准备好，那些内部检查就会发出各种相应的通知。（出于同样的理由，苹果公司 Macintosh 机器上的程序也是这样组织的。）



这两种风格是根本对立的。如果从 Emacs LISP 里对窗口进行图标化或者对 X 的选取缓冲区做明确的设置(或者是明确地调用一些通常是由鼠标驱动的窗口管理器函数完成的操作),就极有可能陷入一种不正常的程序化交互风格(相对于随机风格);与通过 X 完成这些操作的情况相比,画面可能会因窗口边框的意外消失而变得很难看。

所以要多想想“事件驱动”。尽量做到不依赖于具体的程序运行状态;不要在没有考虑设计冲突的前提下用 Emacs LISP 强制执行 X 操作。Emacs 的“把输入事件绑定到函数上”的设计思想能够帮助定义隐含的全局性事件处理循环;尽量使用这种手段,要把序列化的程序运行状态(特别是那些不可见的)减到最少。



第十五章

Emacs 下的 版本控制

本章内容：

- 版本控制的用途
- 版本控制的有关概念
- VC 对基本操作的辅助作用
- 修改注释的编辑
- VC 命令汇总
- VC 模式的标志
- 使用哪一种版本控制系统
- VC 命令细说
- 对 VC 进行定制
- 对 VC 进行扩展
- VC 的不足之处
- 有效地使用 VC

版本控制的用途

如果你曾经编写过大型的程序或者很长的文档，就肯定遇见过这样的情况：以前做的某些修改到最后反而不适当，更让人着急的是你还不知道怎样才能撤消那些修改并回到以前的某个“好”版本上。或者，你为别人开发了一个程序或文档，可因为不知道怎样才能回到别人使用的那个老版本上，也就没有办法把新的程序补丁或注释集成到它里面去。如果你是某个程序或文档开发团队的一员，也许就需要有一种能够把修订史记载下来的办法，方便日后核查哪些修改应该由哪些人来负责。

这些常见的问题都可以用“版本控制系统（version control system）”来解决。版本控制系统能够把某个文件或者某些文件的修订历史自动记录下来。它允许返回到这个历史记录中的任何一个阶段，并且能方便地制作出关于各版本之间差异比较的报告。

在能够运行 Emacs 大多数版本的 UNIX 机器上，有 3 种广泛应用的版本控制系统——SCCS、RCS 和 CVS。这几种版本控制系统都能很好地完成它们的工作；但

由于历史方面的原因，在很多本应该使用它们的场合却见不到它们的身影。造成这种局面的原因之一是这3种版本控制系统的操作界面都比较复杂和琐碎。而且，因为它们还没有被广泛地集成到编辑器里，所以，如果你想使用它们对文件进行检索、编辑、保存或者修改，就必须经过两个烦琐的过程才能完成工作。

在这一章里，我们将介绍一个被称为“VC”（Version Control，版本控制）的Emacs工具，它是一个能够让你轻松使用这3种版本控制系统的Emacs LISP辅助编辑模式。VC能够执行各种版本控制命令（通过Emacs的子进程功能，与运行邮件处理器模式的方法一样）。VC隐藏绝大多数的面向版本控制系统的接口细节；你只需掌握一个命令，就可以用它来完成那些最基本的版本控制操作。

我们假设你使用的是RCS系统，这是因为：(a)人们普遍认为它比出现较早的SCCS系统更先进。(b)它是一种来源广泛的免费软件，(c)它的安装和使用要比CVS系统简单。因此，在对Emacs的VC功能进行介绍时，我们将主要采用RCS系统的术语，并且会把RCS系统的行为当做典型的用法示例。

版本控制的有关概念

处于版本控制下的每个文件都有一份修订记录，这份记录由一个原始版（initial version）和一系列（有时还会有分支）以此为基础的修订版（revisions）组成。

要想对一个文件进行版本控制，就必须先把它“注册”到版本控制系统上；也就是让版本控制系统把注册的文件内容看做原始版本，然后开始记录它的修订历史（注1）。

如果想对一个已经注册的文件进行修改，就必须“取出（check out）”这个文件；也就是通知版本控制系统准备对它进行修改。SCCS或RCS系统还会给文件加锁；在完成修订工作之前，其他人是无法再取出它的（但仍可查看它的内容）。（CVS系统没有使用这种加锁机制，个中的原因我们在此后的第三个段落里进行了解释。）

在SCCS或RCS等使用加锁机制的版本控制系统里，如果有人已经给某个文件加上

注1： 如果你想用VC来管理某个文件，倒不必非得从VC里对它进行注册。VC对已经存在的文件版本控制树也能工作得很好。

了锁，你就无法再收它。如果别人取出了文件之后又忘记了这码事，锁就会一直加在文件上。你也许想把锁“偷卸”下来，也就是把工作文件的控制权以及别人对文件的修改都转移到自己手里，再由你负责“签入 (check in)”一个经过修改的版本。(如果责任心不强，这样做将很危险。)

在对工作文件 (work file，即由你签出的工作用文件副本) 进行修改的过程中，你可以随时“复原 (revert)”工作文件，即丢弃所做的修改并撤销取出操作。可如果你想把自己对文件的修改保存起来，就必须“签入”那些改动。这将使它们成为该文件修订记录中一个永久性的修订版。在RCS和SCCS系统下，签入操作还将释放工作文件上的锁；这样，别人就可以签出这个文件并对之进行编辑。在CVS系统下，文件是不会加上锁的；CVS系统会尽量让你所做的修改与你取出这份文件期间别人所做的修改和平共处，如果它发现其中有所冲突，就会向你求助（即要求由你来进行人工协调）。

注册、签出、复原、签入等操作是版本控制系统最基本的操作。还可以利用版本控制系统做一些其他的事情：可以检索出任何一份已经记录在案的修订版、可以生成一份关于任意两个修订版之间或者关于任何一个修订版与自己（可能已经被改动过的）工作文件之间的差异报告，甚至能够把不想保留的某个修订版删除掉（虽然这种情况非常少见）。

大多数版本控制系统（包括SCCS、RCS和CVS这三个在内）能够给每一份修订版加上一些“修改注释 (change comments)”。也就是说，在签入一个已被注册过的文件时，还可以在修改记录里写上一些关于所做的修改的解释，这些解释不会成为文件本身的一部分。每个修订版都有一个“修订号 (revision number)”，其作用是把它自己在修改记录中的位置标识出来。SCCS、RCS和CVS的修订号都是从1.1开始的。如果文件的修改历史呈线性发展（小型项目修订历史的典型结构），修订号就是一个由两个数字组成的序列，两个数字之间用小数点隔开。

但文件的修改历史完全可能出现旁支，文件的变体版本可以平行地发展。对于这种情况，文件修改历史的主干仍由两段 (two-field) 修订号组成，但旁支的修订号就需要用更多的数字来表示。SCCS、RCS和CVS在旁支的具体命名规则方面各有各的做法；如果想详细了解这方面的资料，请自行查阅版本控制系统的有关文档。版本控制系统还有很多我们来不及在这里介绍的功能。O'Reilly 出版公司出版过一

本关于版本控制系统的好书《Applying RCS and SCCS》，它的作者是 Don Bolinger 和 Tan Bronson。

VC 对基本操作的辅助作用

由于历史上的原因，如果你想进行一些基本的版本控制操作（即注册、签出、签入、复原），就不得不掌握三个或四个不同的 shell 命令，而且这些操作还都必须在你的编辑器之外（最好的情况是在编辑器的一个 shell 子进程里）进行。整个过程既复杂又繁琐，许多人就是因为这个原因才不使用版本控制系统的。

VC 的操作界面就简单得多。这种简单性来源于这样一个事实：不管你的版本受控文件处于什么状态，合乎逻辑的后续操作通常就只有一种。下面是有关的规则：

- 如果你的文件还没有被置于版本控制之下，那么接下来的逻辑步骤就是注册它并签出一份允许你修改的副本。
- 如果文件被注册过但还没有人取出它，接下来的逻辑步骤通常就是签出它并对它进行修改。
- 如果你已经把文件签出，接下来的逻辑步骤就是签入它。
- 如果已经有别人签出了文件，除了等待、你还能做的事情就只有卸下锁了（锁的主人会收到一个锁已被你卸下的通知）。

事实上，VC 模式只有一个基本命令 “**C-x v v**”（命令名是 **vc-next-action**），它的含义是“对这个文件做下一个合乎逻辑的操作”；说得更精确点，“把当前访问的文件带到下一个正常的版本控制状态”。整个流程将沿图 15-1 里的箭头方向运转，这个图给出了正常的版本控制流程（注 2）。这个命令在 Emacs 19 的各个版本里都能使用；调用它的时候，它会自动加载 VC 模式的其他功能并完成它的工作。

注 2： 小技巧：在一个新的工作文件上，发出的第一个 **vc-next-action** 命令通常会从“尚未注册”状态进入“已注册、未加锁”状态，再到“已加锁、可编辑”状态——如果只想对文件进行注册但不打算签出它（这是相当典型的情况），就得用第二条命令从“已加锁、可编辑”状态退回到“已注册、未加锁”状态。这未免有些费力不讨好。如果只想注册不想编辑，可以使用 “**C-x v i**” 命令（命令名是 **vc-register**）。

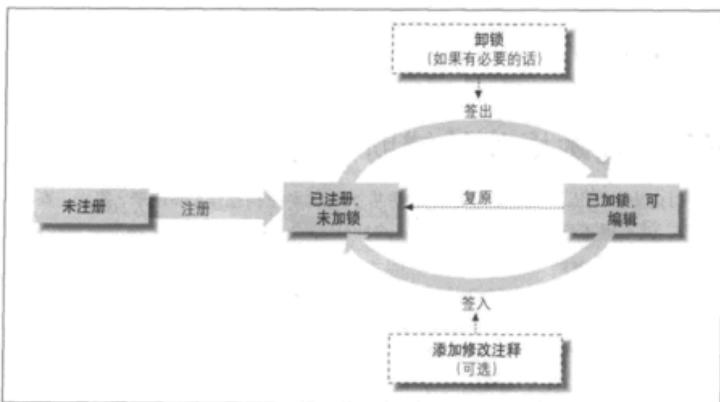


图 15-1：正常的版本控制流程

事情当然没有这么简单。归还某个文件的一批修改时，VC会弹出一个编辑缓冲区以供输入一条修改注释。类似地，卸下一个锁时，VC会弹出一个编辑缓冲区要求提供一个解释；此解释将电邮给锁的主人。

VC 模式还提供一个复原操作：“**C-x v u**”（命令名是 **vc-revert-buffer**）。在实际操作中，负责实现 **vc-next-action** 命令的函数会先检查编辑缓冲区自上次签出之后是否被修改过。如果没有修改过，那么它只复原编辑缓冲区并卸下工作文件上的锁而不会签入一个空白的修改。

修改注释的编辑

在 VC 模式里，有 3 个操作会弹出一个编辑缓冲区以接受注释或通知文本，这 3 个操作是签入、卸锁和文件注册（只发生在某些特定的场合；详见本章后面内容里的解释）。在每一种情况里，有关操作会暂停下来，等你在注释编辑缓冲区中按下“**C-c C-c**”组合键之后再继续完成。可以立即开始输入注释以尽快完成有关操作，也可以去先做点别的事情。VC 不关心程序运行状态（还记得我们在第十四章对程序化接口和非程序化接口的讨论吗？）；它会耐心地等你做完其他事情后再回来完成注释编辑缓冲区中的工作。如果删除弹出来的编辑缓冲区，有关操作就会无声无息地结束。

注释编辑缓冲区是一个明文编辑缓冲区。但是，每打开一个注释编辑缓冲区，其内容就将被保存到注释编辑缓冲区环的一个新位置。可以用“**ESC p**”组合键沿这个环上溯、用“**ESC n**”组合键下溯、用“**ESC r**”沿这个环向回查找文本，或者用“**ESC s**”向前查找文本。根据设计，这些命令与漫游 Emacs 辅助输入区的命令历史记录时所使用的组合键是一样的。这些命令中最常用的是“**ESC p**”。能够调出和编辑上一条修改注释通常很有用。

VC 命令汇总

为了让大家对 VC 模式的其他功能有一个基本印象，这里把 VC 命令汇总在表 15-1 里。我们对每个命令都做了详细的解释；其实，从这些命令的名称上就能把它们的功用猜出来。

表 15-1：VC 命令速查表

键盘操作	命令名称	动作
C-x v v	vc-next-action	前进到下一个合乎逻辑的版本控制状态
C-x v d	vc-directory	列出某个子目录下所有的已注册文件
C-x v =	vc-diff	生成一份版本差异报告
C-x v u	vc-revert-buffer	丢弃上次签入修订版之后的一切修改
C-x v ~	vc-version-other-window	在另外一个窗口打开指定的修订版
C-x v l	vc-print-log	显示某个文件的修改注释和修改记录
C-x v i	vc-register	把文件注册到版本控制系统
C-x v h	vc-insert-headers	给文件加上供版本控制系统使用的标题块
C-x v r	vc-retrieve-snapshot	签出一个已命名的项目快照
C-x v s	vc-create-snapshot	创建一个项目快照并给它起个名字
C-x v c	vc-cancel-version	丢弃一个已经保存起来的修订版
C-x v a	vc-update-change-log	刷新一个 GNU 格式的 ChangeLog 文件

这些命令大致按使用频率由高到低的顺序排列在表格里。这也是我们将在以后的章节里对它们进行讨论的顺序。VC 命令都有相同的前缀“**C-x v**”。你的手指很快就会习惯于这个前缀，所以只要记住那些单个字符的命令后缀即可。VC 模式还有两个辅助性的命令 **vc-rename-file** 和 **vc-clear-context**，它们都没有对应的按键绑定。我们稍后再对这两个命令进行介绍。

VC 模式的标志

如果在某个编辑缓冲区里打开的是一个注册过的文件，VC模式就会占用其状态行的一点空间以给出该文件的版本控制状态。我们将会看到，如果编辑缓冲区正访问一个版本受控文件，模式标签部分（状态行上的括号部分）将给出版本控制系统的名称和文件的修订号。

如果这两部分是用一个短划线隔开的，就说明文件还没有被签出；如果它们是用一个冒号隔开的，就说明文件已经被签出，而显示在状态行上的修订号是签出该文件时它所拥有的修订编号。

如果没有看到这些标志，就说明文件尚未注册。

使用哪一种版本控制系统

我们前面讲过，VC模式至少可以使用3种版本控制系统（今后可能还会增加更多的版本控制系统）。对一个给定的文件来说，VC将根据与之对应的“主控文件（master file）”来选择一种版本控制系统来打开它：主控文件就是保存修改记录的那个文件。

如果使用的是RCS系统，那么每个项目的目录通常会有一个子目录，RCS上控制文件就在这个子目录里；如果使用的是SCCS系统，那么就会是些SCCS子目录。CVS系统的情况有点特殊：每个项目的目录有一个CVS子目录，控制信息就保存在其中，但CVS主控文件通常都集中保存在某个目录里，这个位置通常由环境变量**CVSROOT**给定。

如果VC在这些专用目录里找不到它需要的主控文件，就会到工作文件所在的目录里去查找（因此，如果你不介意自己的工作目录里充斥着各种主控文件，就不必专门创建SCCS或RCS子目录）。VC将检查各种可能性（因此，你完全可以在同一个目录里同时使用多个不同的版本控制系统；但我们并不推荐这种做法）。

如果VC始终没有找到它需要的主控文件，就会依次对RCS目录、SCCS目录，最后是CVS中央目录进行查找。如果它在RCS目录里找到了主控文件，它就会把新文件注册到RCS版本控制系统里，以此类推。如果它在这些目录里也没有找到主控

文件，而你又在让它注册一个文件，它就会认为你打算使用的是 RCS 系统并创建出相应的主控制文件和你的工作文件。

如果不清楚 SCCS、RCS、CVS 等版本控制系统是否能够在你的系统上使用，你可以不带参数地分别执行一次 **comb**、**rcs** 和 **cvs** 命令。如果看到的是出错信息或使用方法信息，就说明对应的版本控制系统能够在你的系统上使用；如果看到的是“command not found（命令未找到）”，就说明对应的版本控制系统不能在你的系统上使用。

VC 命令细说

vc-next-action 是最基本的 VC 命令，它的用途我们已经介绍过了。现在，我们将对 VC 模式的其他命令做详细的介绍。我们选择的讨论顺序是把比较常用和比较简单的命令放在前面，把比较罕见和比较复杂的命令放在后面。

根据你的情况，只要你认为已经学到了足够的东西，就可以随时跳到本章的结尾。不过，虽然越靠后面的命令越不常用，但有关的讨论没准能启发你想出一些记录和组织项目文件的新点子，谦虚好学总是件好事。

文件组和子树

一般说来，打算用版本控制系统进行管理的项目肯定不会只包含一个文件；比较常见的做法是把某个目录及其子目录里的所有文件都交给版本控制系统去管理。所以，如果能够查看到当前工作目录下所有版本受控文件的清单，就能给工作带来很多便利。要是还能把它们当做一个整体来操作就更有好处了。

这些想法都能用 VC 模式实现。可以用命令 “**C-x v d**”（命令名是 **vc-directory**）打开一个编辑缓冲区对目录进行编辑，这个编辑缓冲区运行在一种定制出来的 **Dired**（directory editing，目录编辑）模式下。这个编辑缓冲区不仅列出当前目录下的全部已注册文件，还提供（如果有）哪些文件已被签出、谁是签出人等信息。从 Emacs 19.30 开始，签出、签入操作对文件状态的影响会立刻在这个清单里反应出来。

如果先在这个 **Dired** 编辑缓冲区里（用正常的 **Dired** 文本块选取命令）选取了多个文件，然后执行 **vc-next-action** 或 **vc-revert-buffer** 命令，VC 模式就会对选取的全

体文件执行这个操作。这种做法最适合用在需要同时签入多个文件的场合，而且还有另外一个好处：VC将只弹出一个编辑缓冲区以输入一次修改注释，输入的注释将施用到签入操作创建的每一个修订版。

但 **vc-revert-buffer** 命令相对要保守一些：它会为每个文件提问是否真的想丢弃所做的修改。

从 19.30 开始，有些 Dired 命令在 VC 模式下的 Dired 里被绑定为一些与版本控制有关的操作。比如说，VC 模式下的 “=” 命令进行的是 VC 模式的 **diff** 操作而不再是 Dired 的 **diff** 操作；而 “g” 命令刷新的是目录里的全部 VC 状态 [x]。

差异报告

我们前面讲过，版本控制系统可以生成版本之间的差异报告。VC 模式用来完成这一工作的命令是 “**C-x v =**”（命令名是 **vc-diff**）。这个命令的常见用法是查看当前工作文件与上一次签入的修订版之间都有哪些不同：让你对自己将会签入哪些修改做到心中有数。

如果你给这个命令加上一个前缀参数，即按下 “**C-u C-x v =**” 组合键，那么它将提示输入一个文件名和两个修订号，然后把该文件那两个修订号之间的差异查出来并报告给你。如果较早的修订号是空的（即在 VC 提示输入它时直接按下了回车键），它将默认使用最近一次签入的修订版；如果较晚的修订号是空的，它将默认使用当前的工作文件。

VC 还能给出整个项目树文件的差异报告。如果在 “**C-u C-x v =**” 命令提示输入文件名的时候输入了一个目录名，那么 VC 将把该目录里全部已注册文件的指定版本之间的差异都报告出来。

根据设计要求，VC 生成的差异报告可以通过 Larry Wall 开发的 **patch** 软件工具（现代的 UNIX 操作系统里都有这个工具）转换为一个补丁包。如果你所在的软件合作项目主要以电子邮件为联结手段，这项功能就将是一个莫大的帮助：你可以下载源代码，注册它，进行修改，然后只需一条命令——把所做的修改转换为一个完整的补丁包，再利用电子邮件把它发给其他合作者。

VC给出的差异报告其实会随版本控制系统的不同而变化，因为VC使用了每个系统自己的差异报告器组件，所以这些报告的具体格式会因版本控制系统的不同而有所变化（注3）。但总的说来，这些报告与UNIX命令 **diff** 的输出格式都比较相似。在本章稍后的内容里，我们将介绍如何对差异报告进行定制。

早期版本的检索

“**C-x v ~**”命令（命令名是 **vc-version-other-window**）能够把指定文件的任何一份早期修订版检索出来。修订版将打开这样一个工作文件：工作文件的文件名与注册文件名相同，文件名后缀则给出它的修订号（后缀名的格式是一个英文句号“.”、一个波浪号“~”、修订号，然后又一个波浪号“~”）。换句话说，可以同时打开多个修订版，它们彼此之间不会发生混淆。如果不满足于仅看到差异报告给出的比较结果，可以用这个命令查看早期修订版的全文。

如果你曾设置过 Emacs LISP 的全局变量 **version-control**（VC 出现之后，还使用这个变量的人已经很少了），就会激活 Emacs 的编号备份功能；而 Emacs 的编号备份功能生成的后缀名与 VC 修订版工作文件的后缀格式是很接近的。两者之间的区别是 **version-control** 保存的备份其后缀名只是一个从 1 开始顺序递增的数字，而修订号的中间却有英文句号等其他字符。

查看修改记录

如果在一个已注册文件上使用 “**C-x v l**”（命令名是 **vc-print-log**）命令，VC 就会弹出一个内容为该文件修改记录的编辑缓冲区。可以用这个命令查看文件各修订版的修改注释。

注册文件

可以用 “**C-x v v**”（命令名是 **vc-next-action**）命令把文件注册到版本控制系统里，但这个命令会在注册的同时取出一份可编辑的副本。但有时候只想对文件进行注册而不想签出它，这时就需要使用 “**C-x v i**” 命令（命令名是 **vc-register**）。

注 3：这个问题稍微有些复杂。事实上，为了让 SCCS 系统的 **scsdiff** 命令在调用次序上与 RCS 系统的 **rcsdiff** 命令保持一致，VC 用自己的一个脚本程序对前者进行了打包。

插入版本控制块

大多数版本控制系统都鼓励你在自己的文件里嵌入一些能够在签出、签入以及复原操作中自动更新的字符串。这些字符串的作用是在文件里自动插入一些信息，比如文件的当前修订号，最后一次修改文件的人，以及文件的最后一次签入时间等。

文件内容里的版本控制字符串大部分照抄自VC放在状态行上的版本信息，剩下的则来自“C-x v l”命令（命令名是**vc-print-log**）。这个功能看起来似乎没有什么用处，但嵌在文件里的版本控制字符串能够出现在经编译得到的程序代码里，对今后的调试和升级会有帮助。

此外，还可以让VC根据嵌在文件里的版本控制字符串而不是根据主控文件分析该文件的版本信息（如果主控制文件很大，这种做法可以加快打开文件的速度）。我们将在后面讨论VC模式定制办法的有关章节里介绍如何强制实现这种操作。

另外一种情况是，可能需要经常使用Emacs以外的其他软件查看版本受控文件，而在其他软件里是看不到显示版本控制信息的Emacs状态行的。对于这种情况，嵌在文件里的版本控制字符串的价值就体现出来了。为了插入这些字符串，VC专门准备了一条命令。

正在访问一个加有锁的可编辑文件的时候，如果按下“C-x v h”组合键（命令名是**vc-insert-headers**），VC就会根据该文件的语法推断出怎样才能把相应的版本控制信息以适当的注释格式插入到文件里并插入这些字符串。VC对C语言代码和nroff/troff/groff代码特别熟悉，而且还经常能根据Emacs各主编辑模式所设置的全局变量**comment-start**和**comment-end**做出正确的推断。如果它无法做出更好的判断，就使用“#-to-\n”这样的注释格式（shell、awk、Perl和tcl等许多UNIX语言的注释使用的都是这种格式）。

C语言代码在插入版本控制信息方面的行为是比较特殊的。在默认的情况下，Emacs不会把C文件的版本控制信息放在程序中的注释里。Emacs会把这些信息保存在一个名为**vcid**的静态字符串变量里。这将使版本控制信息以字符串的形式出现在编译器生成的目标文件里：可以通过C语言的字符串命令查出二进制执行文件是用哪个版本的源文件生成的。

创建和检索项目快照

软件项目往往包含有许多文件，而这些项目文件在开发过程中又会经过多次修订，但并非每次修订都会对全体文件进行修改。这样，在经过一个时期之后，有些文件的修订号已经很大；而有些文件因为不需要经过那么多次的修订，因而修订号还很小。当我们需要把项目交付给顾客或者准备请第三方对项目进行评估的时候，就必须把这些修订号参差不齐的项目文件都放到一个“项目快照（snapshot）”里。换句说话，项目快照就是软件开发项目的阶段性成果。软件的“升级版”或“测试版”其实就是一些项目快照。

随着开发工作的进行，项目文件的修订号会越来越多，也就越来越不好记忆。等到想对项目文件的全体或者一部分做阶段性总结的时候，就会感到无从下手。幸好大多数版本控制系统都已经考虑到这个问题，它们允许给软件项目的阶段性成果起一个符号名称，用这个符号名称把有关的文件全都关联在一起；此后，就可以用这个符号名称来完成检索修订版或者生成差异报告等版本控制工作了。

RCS 和 CVS 本身就有项目快照功能。SCCS 虽然本身不具备项目快照功能，但 VC 模式用自己的代码在 SCCS 系统下进行了仿真。在实际工作中，版本控制系统给项目快照起的符号名称与 VC 给项目快照起的符号名称看不出有什么不同。用 VC 仿真出来的项目快照功能只有一个美中不足：如果在 VC 以外的地方使用 SCCS 系统，那些 SCCS 工具就不会知道项目快照的符号名称到底代表什么。

“**C-x v s**”命令（命令名是 **vc-create-snapshot**）会提示输入一个符号名称。然后，VC 会把当前目录里所有已注册文件的当前修订版都与这个名称关联起来。

对需要用到修订号的其他 VC 命令来说，用 **vc-create-snapshot** 命令创建出来的符号名称将成为它们的合法参数。项目快照对 **vc-diff** 命令来说特别有用，它意味着将能够生成项目快照彼此之间或者与你签出的工作文件之间的差异报告。“**C-x v r**”（命令名是 **vc-retrieve-snapshot**）以一个项目快照的符号名称为输入参数，把当前工作目录里每一个已注册文件关联在该名称下的修订版取出来。

如果当前目录中某个已注册文件被别人签出，两个快照命令就都会执行失败。快照创建命令会报告无法创建快照，而快照检索命令会报告没有检索到文件。**vc-create-snapshot** 命令执行失败的原因是不想创建出一个无法完整恢复当前状态的项目快



照。**vc-retrieve-snapshot**命令执行失败的原因是不想做无用功，因为肯定需要在签入或复原工作文件之后再进行一次检索。

刷新 ChangeLog 文件

自由软件基金会（Free Software Foundation, FSF）在项目管理方面使用的某些记录方法与SCCS、RCS和CVS等版本控制系统有所不同，为了让它们互通有无，VC模式准备了一条“C-x v a”命令（命令名是**vc-update-change-log**）。按照FSF项目开发的要求，每修改一次文件，就应该写下一条修改注释；这些修改注释将按时间顺序记载到一个文件里以备查考。因此，FSF项目的每个目录通常都会有一个名为*ChangeLog*的文件，其内容就是该目录里每个文件的修改注释，而且这些注释都加有时间戳（timestamp）。简单地说，*ChangeLog*文件就是FSF项目的修改历史；而VC虽然也有类似的功能，但那是通过版本控制系统的修改注释功能来实现的。

用不着把每个修改注释输入两次，VC提供的一个挂钩能够把新添加的修改注释从当前目录下的主控文件里复制出来，并把它们以正确的格式追加到*ChangeLog*文件的末尾。

vc-rename-file 命令：重新命名版本受控文件

重新命名版本受控文件是一件比较麻烦的事。在RCS或SCCS系统里，只把工作文件的名字改过来还不够，还必须把它所关联的主控文件的名字也改过来。在CVS系统下，由于一些很难用三言两语解释清楚的原因，要想不弄坏点东西就给文件改了名字可以说根本就不可能。

vc-rename-file试图隐藏这个操作的具体细节，并且能够捕获各种可能发生的错误并发出通知。它的用法很简单：先提示输入旧文件名和新文件名，然后尝试完成操作，如果不成功就发出通知。

警告：重命名操作与SCCS系统下仿真出来的项目快照功能很难协调工作。这也是我们推荐使用RCS和CVS系统的重要理由之一。



vc-clear-context 命令：当 VC 被弄糊涂时

需要根据文件的版本控制状态决定如何继续执行的文件系统操作其开销往往很大，而且速度也会很慢；在 NFS 等网络化的环境里，这一问题尤其突出。如果用 VC 来做这些事，情况也差不多（除非你不让它这么做，我们马上就会看到）。

它有两个主要的方法：(1) 把每个文件的信息（比如锁的主人或修订号等）都放到内存中的缓存里，不采用每次都得用运行版本控制工具把它们从对应的主控文件里分析出来的办法；(2) 设法从某个已注册文件的写权限里推导出它的版本控制状态。简单地说，如果某个已注册文件的写权限是“可写”，VC 就将认为它正处于“已签出并已加锁”状态；如果某个已注册文件的写权限是“不可写”，它就不是当前编辑的取出版本。

在多用户环境里，VC 缓存的信息和它根据文件访问权限做出的判断往往会让它走上歧途。因为经常有人以手动的方式不经过 VC 就改变文件的权限，所以这种情况常常发生。

如果认为发生了这样的情况，就应该立刻使用 **vc-clear-context** 命令。这个命令将强制 VC 丢弃内存中与当前工作文件的版本控制状态有关的缓存信息。

此外，在理论上还存在着这样一种可能：因为出现了竞争现象（race condition）而把 VC 弄得不知所措，这种竞争可能出现在两个或者更多个 VC 之间，也可能出现在 VC 与别人单独运行的 SCCS、RCS 或 CVS 工具程序之间。这种问题不仅 VC 会遇到，两个或者更多单独运行的工具程序也有可能出现竞争现象（虽然概率不大）。不过，这种竞争现象即使是 VC 也很少遇到；在笔者多年的程序设计工作中，从没听说发生过这样的事情。

如果你对这个问题感兴趣，请查看 VC 的源代码（Emacs LISP 源代码目录中的 *vc.el* 文件）里的一段注释，这段注释对多用户环境下的冲突和竞争现象做了细致深入的分析。但大家应该有充分的信心，VC 是相当成熟和强大的。

对 VC 进行定制

作为 Emacs 的一种辅助编辑模式，VC 的行为方式有很多是可以定制的，这需要我们对某些属于 VC 模式的 Emacs 变量进行设置。我们将介绍几个最重要的。

vc-default-back-end

这个变量负责设定 VC 缺省使用的版本控制系统，如果它找不到想签入的文件的主控文件，就会使用这个变量设置的版本控制系统进行处理。它的缺省设置是“RCS”，其他合法值是“CVS”和“SCCS”。如果你手里只有 SCCS 系统或者你的站点要求必须使用 CVS 系统，就需要修改这个变量。

vc-display-status

如果这个变量的取值不是“nil”，就会在编辑缓冲区的状态行上显示出文件的修订号和版本控制状态。如果你正在通过很慢的网络链接运行 VC，就可以考虑把这个开关变量关掉以减少开销巨大的主控文件查询操作。

vc-header-alist

这个变量的作用是通过一个关联列表把版本控制系统后端映射到由 **vc-insert-headers** 命令插入的字符串上。它的缺省值可以在 VC 源代码里查到；如果想让自己的版本号标题使用另外一种格式，可以修改这个变量。

vc-keep-workfiles

在正常情况下，VC 每执行一次签入操作就会留下工作文件的一个只读副本。这个功能是很有用的，因为 **make** 和其他工具能够在预期的地方找到工作文件。要是硬盘空间很紧张，就可以考虑关掉这个变量；但以后每当有 VC 以外的其他工具需要用到这个工作文件时，就不得不明确地执行一次签出操作。（Emacs能够通过一小段长驻内存的 VC 代码掌握版本控制方面的情况；它能在必要时自动执行取出命令，取出一个只读副本，并且不用加锁。）

vc-mistrust-permissions

这个变量一般设置为“nil”。把它设置为“t”等于告诉 VC 不要根据文件的访问权限或属主权限判断它的版本控制状态。这个改变会大大降低 VC 的执行速度，但如果（比如说）开发工作正在多个不同的目录里同时展开并且工作文件的访问通过符号链接（symbolic link）来实现，可能就需要把这个变量设置为“t”。此后，链接的访问权限和属主权限与工作文件的版本控制状态将没有任何关系。

vc-suppress-confirm

这个变量的缺省值是“nil”。如果它的取值不是“nil”，以后再使用 **vc-revert-buffer** 复原命令丢弃所做的修改时，VC 就不再提示进行确认。

vc-initial-comment

大多数版本控制系统允许（但不要求）对文件进行注册操作时输入一条初始注释，也就是修改记录中的第一句话。如果这个变量的取值不是“nil”，VC将在注册过程中弹出一个编辑缓冲区以输入这条注释，就像它在签入操作中要求输入修改注释时的做法一样。

diff-switches

Emacs 的 *diff.el* 模式将在需要生成修改报告的时候把这个全局变量里的命令行开关传递到 UNIX 操作系统的 **diff** 命令。VC 也以同样的方式使用这个变量。这个变量的缺省值是单个开关 “**-c**”，即使用 **context-diff** 格式；用来设定使用 **unified-diff** 格式的开关 “**-u**” 也比较常见。

vc-consult-headers

如果这个变量（通常是 “nil”）设置为 “t”，VC 就将从嵌入在每个文件里的版本控制字符串处获取文件的状态和版本编号信息，不再查看对应的主控文件。（如果主控文件很大，这种设置将提高 VC 打开文件的速度，但在激活这项功能之前一定要保证所有的文件都包含正确的版本控制信息块）。

Emacs 的在线帮助系统对另外一些不这么重要的全局变量做了详细的说明。

对 VC 进行扩展

VC 的设计思想从一开始就是让它成为多种版本控制系统的一个操作前端。为了使接插新系统的工作容易进行，人们对实际运行版本控制工具的代码与用户级组件逻辑进行了仔细的隔离。VC 的作者最初只写了它与 SCCS 和 RCS 的接口，对 CVS 系统的支持是由另外一个人后来添加进去的，其难度也确实不大。

在我们编写这本书的时候，听说有人正在编写 VC 到 ClearCase 的接口；ClearCase 是一种流行的商业化项目控制系统。考虑到自由软件基金会对非自由软件的敌视态度，那些代码能否被接受到 GNU Emacs 的发行版本里还是一个未知数。不过，即使自由软件基金会不接受 ClearCase 所做的修改，它们也会通过别的渠道逐渐进入 VC 的覆盖范围。



本书只介绍了3种版本控制系统；但等大家读到这本书时，你的VC可能已经能够很好地处理这3种以外的其他版本控制系统了（但在可预见的未来，这3种版本控制系统却极可能仍然是因特网/UNIX世界里最流行的）。如果你是一位熟练的EmacS LISP程序员（或者如果你想成为一个这样的人）并拥有一个你最喜欢的版本控制系统，请钻研它的源代码并对VC进行扩展，让VC也能够使用你的版本控制系统，然后把你的成果拿出来与大家分享。

VC的不足之处

VC决不是项目管理问题的终极解决方案。它能够对独立作者的程序设计或文档管理工作提供极大的帮助，也能够对有多名程序员参加的中小型项目提供重要的帮助。可大型项目往往涉及到许多组件和许多目录，单凭VC本身的能力是无法胜任的。当用它来管理大型项目时，它的某些不足就会显露出来，其中包括：

- 它没有集成修改请求或问题报告系统。不能及时发现潜在的需要修改的地方。
- 它对项目文件的组织管理手段仅限于目录子树。这种局限性在面对涉及到多个目录的大型项目时可能会出现一些问题，特别是当两个或者更多目录需要共享一个开发库或者子树的时候。

这些不足在小型项目上很容易得到解决，比如说，变体版本可以用编译条件来对付，比如C语言中的“`#ifdefs`”语句；修改请求可以单独保存在某个数据库（比如FSF的GNATS系统）里。程序员可以把需要进行重命名操作的开发阶段条理清晰地记在脑子里。

随着项目在规模和复杂程度上的增加，还想用这些小手段来解决所有的有害的冲突情况就不那么容易了；而小问题积累多了，就会造成致命的损害。要想对超大规模项目的复杂性加以控制，就需要有一个基础更强大（因而约束性和复杂性也更大，这是没有办法的事）、功能比版本控制系统更多的支持环境，即一个完备的项目管理系统。

如果想了解更多关于项目管理系统的设计问题，请阅读《Applying SCCS and RCS》一书的后半部分。

有效地使用 VC

最后，我们希望有版本控制经验的读者留意这句话：要想有效地使用 VC，就一定要尽早和及时地签入所做修改。

如果你习惯于笨重而又难用的版本控制接口，比如独立的 SCCS、RCS 或 CVS，那么你过去养成的习惯可能会让你无法充分地享受到 VC 的好处。你可能不习惯经常存盘；不习惯经常查看某个文件子树的状态报告；也常常想不起利用项目快照之类的功能来总结阶段性的成果。

多花点时间和精力对自己进行再教育是值得的。你将发现，VC 决不是一种无关紧要的辅助工具，VC 下的版本控制是一种极大的解放。如果你经常签入所做修改，就会发现自己将能承受更多探索性质的尝试，因为你有把握在必要时能够迅速返回到一个已知的良好状态。

不管是在项目快照之间还是在项目快照与修改的工作文件之间，用 VC 给程序打补丁能够做到又快又完整；当认识到这一点时，你对分布式开发的想法肯定会有所转变。松散型的开发团队往往会因沟通不足而降低效率，尤其是当程序员各自保有一份源代码的本地副本而主要的通信手段又仅仅是电子邮件时将更是如此；VC 提供的打补丁功能将大大改善这类开发团队的工作情况。经验表明，虽然 VC 还远不是这类沟通问题的一个完美解决方案，但与依靠个人技术完成小修小补的旧时代相比，这是前进道路上的重要一步。



第十六章

在线帮助

本章内容：

- Emacs 的自动补足功能
- 帮助命令
- 针对复杂 Emacs 命令的帮助功能

就文本编辑器而言，Emacs 的在线帮助功能是最全面的；即使与其他种类的程序相比，它的帮助功能也算得上是最好的。如果从规模上比较，编写 Emacs 在线帮助文档所花费的时间肯定比我们编写这本书所花费的时间要长。Emacs 的在线帮助功能主要由 3 个部分组成：

- 自动补足功能——Emacs 能帮助完成事物名称（比如文件名）的输入工作。
- 帮助键（通常是“C-h”）——它可以查看关于各种问题的帮助，还能让你使用 info 数据库和 Emacs 教程等工具。
- 针对复杂命令的帮助功能——比如 query-replace 和 direct 等命令。

Emacs 的自动补足功能

我们在第一章就见识过 Emacs 的自动补足功能。Emacs 的自动补足功能远不止是一项功能：应该说它是 Emacs 设计工作中的一项基本原则。我们可以把它表述如下：

如果需要输入某个事物的名称，而这个名称是有限个数的可能性中的一种，Emacs 就能在输入可能最少个数的字符后判断出想输入的是什么。

换句话说，只需输入一个“最短无二义前缀”就足够了，名称的剩余部分由 Emacs

来补足。所谓“最短无二义前缀”的含义是“从名称中的第一个字符开始，到足以把该名称与其他可能性区分开来的字符”。Emacs比较重要的几样事物都有从有限个数的可能性中挑选出来的名称，其中包括：

- 命令
- 某给定目录里的文件
- 编辑缓冲区
- Emacs 变量

当Emacs提示需要在辅助输入区里输入某个事物的名称时，几乎总是可以利用它的自动补足功能。在输入事物名称的过程中，如果想让Emacs来帮助补足它，那么有3个字符可供选用，它们是制表符（TAB）、空格（SPACE）和问号（？）。这3个字符的作用是：

TAB 尽可能多地补足这个名称

SPACE 补足这个名称直到下一个标点符号

? 以此前输入的字符为依据，把可用选择列在“*Completions*”窗口里

这3个字符里最有用的是**TAB**。

举个常见的例子。假设已经按下“**C-x C-f**”组合键以访问一个文件，准备访问的文件是一个名为“*program.c*”的C语言程序。输入“**pro**”之后按下**TAB**键；Emacs会自动把这个文件名补足为“*program.c*”。如果按下的是空格键，Emacs就只补足为“*program*”。在Emacs补足这个文件名之后，按下回车键后就可以打开这个文件。

在使用自动补足功能前需要输入多少个字符呢？这要看当时的其他可能性情况。如果“*program.c*”是目录里唯一的文件，输入字母“**p**”后马上按**TAB**键都行（注1）。如果目录里还有其他文件，可它们的名字都不是以字母“**p**”开头的，也可以像刚才那样做。但要是还有一个名为“*problem.c*”的文件，就只能先输入“**prog**”后才能按下**TAB**键；“**prog**”正是此例中最短的无二义前缀。如果只输入“**pro**”就按下**TAB**

注1：不能没输入字母“**p**”就按**TAB**键，这是因为当前目录和它的父目录也是文件名候选，它们的名字分别是“.”和“..”。可如果你访问的文件是一个目录，Emacs就会运行**dired**命令。

键，Emacs 将打开一个“*Completions*”窗口，里面是供自动补足功能选取的各种可能性——就此例而言就是“*program.c*”和“*problem.c*”，然后，Emacs 会把光标放回到辅助输入区里以便完成文件名的输入工作。如果按下的是问号而不是 TAB 键，情况将与刚才一样。此时，再输入一个字母“g”，然后按 TAB 键；Emacs 就能自动补足出文件名“*program.c*”了。

再来看一个例子。假设你为自己的 C 语言程序编写了一份文档，文档存放在文件“*program.doc*”里；而你现在想访问它。按下“C-x C-f”组合键，输入“**prog**”后按下 TAB 键。Emacs 自动补足出“*program.*”（注意里面的句点）。此时，可以再输入一个字母“d”并按下 TAB 键；Emacs 将自动补足出完整的“*program.doc*”。换句话说，在输入事物名称时可以反复使用自动补足功能。

最后，假设你的目录里还有一个就叫做“*program*”的文件，它是你的 C 文件经过编译得到的结果，但你现在想访问的仍是文档文件。于是，输入“**prog**”后按下 TAB 键；Emacs 自动补足出“*program*”。从这里开始，TAB 键和空格键做的事就不一样了。如果再次按下 TAB 键，Emacs 将在辅助输入区里显示一条信息 “[Complete, but not unique]”（自动补足，但不惟一）；但如果按下的是空格键，Emacs 会认为你对文件“*program*”不感兴趣并将继续进行自动补足。因为你有两个分别叫做“*program.c*”和“*program.doc*”的文件，而此时 Emacs 只补足出了“*program.*”，所以你必须再敲入字母“d”并按下 TAB 键才能打开“*program.doc*”。

自动补足功能也同样适用于编辑缓冲区的名字，比如按下“C-x b”组合键以切换到当前窗口另一个编辑缓冲区的时候。它还能用来为“ESC x”组合键自动补足 Emacs 的命令名——但此时它将多出一项功能。当你准备输入的是一个文件名或者一个编辑缓冲区名时，你想打开的文件/编辑缓冲区可能还不存在（比如想创建一个新文件的时候）。如果是这种情况，当然得由你本人来输入文件名或者编辑缓冲区名，然后按回车键。但当你准备用“ESC x”组合键来执行 Emacs 编辑命令时，这个命令肯定是已经存在的。因此，即使按的是回车键，Emacs 也将开始对命令名进行自动补足。

比如说，如果想把某个文本文件的编辑缓冲区放到自动换行模式（参见第二章）里，就可以在输入“**ESC x auto-f**”之后按下回车键，不必输入完整的“**ESC x auto-fill-mode**”。如果输入的命令前缀不惟一（有二义性）——假设输入的是“**ESC x aut**”并按下了回车键，则这个回车键的作用就相当于 TAB 键；就这个例子而言，

它将自动补足出“**a u t o -**”。如果再按一次回车键，Emacs 将打开一个“*Completions*”窗口列出可用的选择，也就是“**auto-fill-mode**”和“**auto-save-mode**”。如果想执行的命令是“**auto-fill-mode**”，就需要再输入一个字母“f”并再次按下回车键。

针对命令名的回车键自动补足功能会给工作带来很大的方便。在用过 Emacs 一段时间之后，你对自己常用命令的最短无二义前缀就应该比较熟悉了；只需输入这些前缀而不是完整的命令名将大大减少你的输入工作量（注 2）。

Emacs 还能自动补足 Emacs 变量的名字。在第二章和其他一些章节里，我们已经见过很多用“**ESC x set-variable**”命令来改变 Emacs 变量值的例子。而刚才介绍的回车键自动补足功能不仅适用于命令名，也同样适用于变量名；这就是说，在“**ESC x set-variable**”操作中也能使用自动补足功能。事实上，命令和变量都是特殊形式的 Emacs LISP 符号，而 Emacs 能够用回车键对一切类型的 LISP 符号进行自动补足。当你需要在工作中使用本章介绍的那些帮助命令时，LISP 符号上的自动补足功能将给你带来很大的方便。

对自动补足功能进行定制

如果你读过第十一章并且愿意尝试着对 Emacs 变量进行设置，就应该知道有几个变量是可以用来定制 Emacs 的自动补足功能的。变量 **completion-auto-help** 将决定是否要在按下空格键或 TAB 键而此前输入的是一个二义性前缀时自动打开一个“*Completions*”窗口。它的缺省值是“t”，意思是自动打开这个窗口。如果把它设置为“nil”，Emacs 就不会自动打开一个“*Completions*”窗口，它将在辅助输入区（minibuffer）里显示 “[Next char not unique] (下一个字符不惟一)” 信息几秒钟。

如果你是一位程序员或者你使用 TeX 等文本排版软件，你创建的就将是一些不适合人类阅读的文件，比如编译器生成的目标文件或者文本排版软件生成的打印文件等。一般说来，当你使用 Emacs 的自动补足功能时，你是不想让 Emacs 把这类文件也列为候选的；比如说，如果你有两个文件“*program.c*”和“*program.o*”（编译器输出的目标代码），那么你通常只想让 Emacs 列出前者做为候选。Emacs 也确实提供了

注 2：举个例子。如果你需要经常修改你的.emacs 文件，就可以用“**ESC x eval-c**”前缀做为“**ESC x eval-current-buffer**”命令的替代品。

这样的功能：事实上，你可能已经注意到这种情况了：如果输入“*program*”并按下 TAB 键，Emacs 将忽略“*program.o*”而只补足出“*program.c*”。这种行为是由变量 **completion-ignored-extensions** 控制的；它是一个由文件名后缀构成的列表，Emacs 在补足文件名时将不把以它们为后缀的文件名列为准。在默认的情况下，这个列表收录着 Emacs 备份文件使用的波浪符(~)、程序员使用的“.o”、TeX 用户使用的一系列文件名后缀、Emacs LISP 程序员使用的“.elc”（经过字节编译的 Emacs LISP 代码），以及其他一些文件名后缀。（当然，如果你确实想查看这些文件，可以亲手输入它们完整的文件名。）

如果想把自己的“忽略”文件名后缀添加到这个列表里，就需要把一条下面这样的语句添加到“*.emacs*”文件里：

```
(setq completion-ignored-extensions
      (cons "suffix" completion-ignored-extensions))
```

比如说，假设你正用一台 PostScript 打印机对文本进行排版，文本排版软件将生成一个后缀名为“.ps”的打印文件。你不想让自动补足功能把这些打印文件也列为候选，所以你必须将下面这条语句添加到“*.emacs*”文件里：

```
(setq completion-ignored-extensions
      (cons ".ps" completion-ignored-extensions))
```

最后，如果把变量 **completion-ignore-case** 设置为“t”（或者任何不是“nil”的值），Emacs 将在使用自动补足功能时忽略字母的大小写情况。这个变量的缺省值是“nil”，意思是 Emacs 将区分字母的大小写。

帮助命令

Emacs 有很多帮助命令，它们有的是标准的 Emacs 命令，有的是“C-h”帮助键下的选项（注 3）。它们可以用来查找关于命令、键盘操作、变量、编辑模式，以及各种 Emacs 事物的信息。最基本的帮助命令是“C-h C-h”（命令名是 **help-for-help**）；这个命令的第二项必须是“C-h”，不管你是否重新绑定过帮助键。“C-h ?”组合键也能用来执行 **help-for-help** 命令。这个命令将使 Emacs 在一个窗口里打开“*Help*”编辑缓冲区，里面列出的是关于各种帮助命令的说明。可以敲入其中的

注 3： 如果你重新绑定帮助键，就需要把本章后面内容里的“C-h”都看做你的新绑定。但也有几个例外，我们会在遇到它们的时候告诉你。

某个帮助键查看相应的帮助信息；如果按的是空格键，“*Help*”窗口将下卷，就好像按的是“C-v”组合键一样。按任意其他键将关闭“*Help*”窗口。如果到达帮助文档的末尾，按一个帮助键或任何其他键都将退出操作。

列在“*Help*”窗口辅助输入区消息里的是这样一些按键：如果在敲入帮助键之后又按下这些键，就会执行相应的Emacs帮助命令。我们将依次对这些按键进行说明。帮助命令又可分为两大类：一类负责提供特定问题的答案，另一类负责给出关于Emacs的概括性资料。

在对Emacs比较熟悉之后，你将发现第一类帮助命令对自己更有帮助。因为它的规模是如此的巨大而功能又是如此的丰富，所以你经常会遇到这样的情况：需要在里面查找某个键盘操作命令或编辑命令的细节、或者想用Emacs完成某项工作，可又不知该如何下手。正如我们在书中反复强调的那样，你想要的东西几乎都能在Emacs里找到，但问题的关键是怎样才能找到它们。帮助命令把这种查找工作变得既快捷又简单，你既不需要离开Emacs，也不必做一名参考手册的奴隶。

详细资料

我们从用来查找某些细节的帮助命令开始讲起。表16-1里是一些最常用的帮助命令。

表16-1：用来查找详细资料的帮助命令

键盘操作	命令名称	回答的问题
C-h c	describe-key-briefly	这个按键组合将运行哪个命令
C-h k	describe-key	这个按键组合将运行哪个命令？这个命令的作用是什么
C-h l	view-lossage	最近输入的100个字符是什么
C-h w	where-is	这个命令的按键绑定是什么
C-h f	describe-function	这个函数的作用是什么
C-h v	describe-variable	这个变量的含义是什么？它有哪些可取值
C-h m	describe-mode	查看当前编辑缓冲区所在编辑模式的有关资料
C-h b	describe-bindings	这个编辑缓冲区都有哪些按键绑定
C-h s	describe-syntax	这个编辑缓冲区使用的是哪个语法表

最需要使用在线帮助功能的场合大概是按错键的时候：编辑缓冲区里出了点问题，可你却不知道下一步该怎么办。通常，最安全的方法是按下“**C-x u**”组合键（命令名是 **undo**）撤销刚才的操作。但有几种情况（例如一个与你愿望背道而驰的 **replace-string** 命令）是这个命令也帮不上忙的。如果你还能想起自己刚敲入的组合键是什么，就可以用“**C-h c**”组合键（命令名是 **describe-key-briefly**）来看看它到底执行了哪个命令；在提示符处再按一次刚才误按的组合键，Emacs 将把绑定到这个组合键上的命令显示在辅助输入区里。如果仅知道命令名还不够，请按下“**C-h k**”组合键（命令名是 **describe-key**），它将弹出一个“*Help*”窗口，里面除命令的名称和按键绑定之外还会给出一个比较详细的说明。要是你想不起自己刚才误按了哪些键，就需要求助于“**C-h l**”组合键（命令名是 **view-lossage**）。这个命令将弹出一个“*Help*”窗口，里面是最近输入的 100 个键；引发错误的（组合）键应该位于比较靠后的位置；接下来，你可以用“**C-h c**”和“**C-h k**”组合键来查出它的功用。

如果你想了解一个没有按键绑定的命令的情况，请按下面的方法操作。按下“**C-h f**”组合键（命令名是 **describe-function**）并在提示符处输入命令的名称；Emacs 将打开一个“*Help*”窗口，里面是关于这条命令的文档。如果你还记得命令名却忘了它的按键绑定，请按下“**C-h w**”组合键（命令名是 **where-is**）。它的作用正好与“**C-h c**”组合键“相反”，如果给定命令有按键绑定，它就会把那个绑定显示在辅助输入区里；如果给定命令没有按键绑定，就在辅助输入区里显示下面这条信息：

```
command-name is not on any keys
```

有时候，你可能忘了某个变量应该如何设置，比如说，哪个值会让 Emacs 的查找操作区分或者忽略字母的大小写（由变量 **case-fold-search** 控制）？我的编辑缓冲区每隔多长时间会自动存盘一次（由变量 **auto-save-interval** 控制）？如果你按下“**C-h v**”组合键（命令名是 **describe-variable**）后再输入那个变量的名字，Emacs 就会把它的可取值和它的文档显示在一个“*Help*”窗口里。“**C-h f**”、“**C-h w**”和“**C-h v**”都允许在输入命令名 / 变量名的时候利用 Emacs 的自动补足功能。**“C-h f”** 和 **“C-h v”** 组合键对 Emacs LISP 程序员也特别有用；特别是 **“C-h f”**，它会把全部函数的资料都查出来，而不仅限于绑定到按键组合上的那些命令。

另一种需要帮助的情况是：你正在某个特殊的编辑模式中进行着工作，比如 **shell** 模式、某种程序设计语言或字处理器的专用编辑模式等，可怎么也想不起它的某个专

用命令或者其他诸如缩进样式之类的细节了。如果在运行某个编辑模式的编辑缓冲区里按下“**C-h m**”组合键（命令名是**describe-mode**），Emacs 就会弹出一个“*Help*”窗口，里面显示着这个编辑模式的文档。编辑模式的文档通常包括以下内容：它的局部按键绑定（即该编辑模式的专用命令和与之关联的按键组合）、它的定制变量以及其他一些有意思的东西。

表16-1中的最后两条帮助命令普通用户很少会用到，但它们对技术娴熟的Emacs定制者却有很大的帮助。“**C-h b**”组合键（命令名是**describe-bindings**）会把活跃在当前编辑缓冲区里的各种按键绑定（包括局部（即该编辑缓冲区专用的）绑定和全局绑定）都显示在一个“*Help*”窗口里；如果你的Emacs是运行在X窗口系统里的，则鼠标动作、菜单项选取等操作的绑定（参见第十四章）也都会显示在这个“*Help*”窗口里。

“**C-h b**”的输出内容相当多。如果只想查看那些有特定前缀的按键绑定的资料，就需要在“**C-h**”的前面输入相应的前缀。比如说，“**C-x C-h**”组合键打开的“*Help*”窗口里将只列出以“**C-x**”开头的全部按键绑定。

“**C-h s**”组合键（命令名是**describe-syntax**）打开的“*Help*”窗口将给出当前编辑缓冲区所使用的语法表（*syntax table*，参见第十二章）的有关说明。

apropos 命令

当你想用Emacs来完成某项工作却又不能肯定应该使用什么命令或者应该设置哪些变量的时候，就需要使用另一个帮助命令。这个命令就是**apropos**，一个在功能和用法上都与很多软件库所使用的基本信息检索系统非常接近的Emacs帮助命令。**apropos**命令有3种形式，我们把它们汇总在表16-2里。

表 16-2: apropos 命令

键盘操作	命令名称	回答的问题
C-h a	command-apropos	这个概念都涉及到哪些命令
(无)	apropos	这个概念都涉及到哪些函数和变量
(无)	super-apropos	哪些函数和变量的文档里提到了这个概念

这3个命令都要求用正则表达式（参见第三章）回答它们的提示。按下“**C-h a**”组合键并给出一个正则表达式时，Emacs会把所有与它匹配的命令都找出来；它会把它们的按键绑定（如果有）和文档的第一行显示在一个“*Help*”窗口里。（Emacs命令和变量的文档基本上都是这样写的：第一行是一句概括，其余部分是详细说明。）如果不愿意使用正则表达式，也可以使用普通的查找字符串代替，但先决条件是不涉及特殊字符。比如说，如果想知道 Emacs 都支持哪些替换命令，请按下“**C-h a**”组合键并输入“**replace**”；Emacs会把关于下面这些命令的资料显示出来：

- **query-replace**
- **query-replace-regexp**
- **replace-buffer-in-windows**
- **replace-regexp**
- **replace-string**
- **tags-query-replace**

如果你曾经使用过某种信息检索系统，就该知道要想用好一个这样的系统必须掌握一些技巧。你必须认真挑选概念（即查找字符串），让它们既不至于太宽泛（以免产生太多无用的输出），也不至于太细致（以免因输出结果太少而找不到想要的资料）。准备使用 **apropos** 命令的时候，这个问题会变得更加突出：**apropos** 命令与 **command-apropos** 原理相同，但信息量更大，它不仅会查找出有关的命令，还会把与此有关的全部函数（包括 Emacs 的内部函数）和全部变量都找出来。**super-apropos** 命令在这个问题上是最突出的，因为它的查找范围在函数和变量之外又增加了它们的文档。

如果输入的查找字符串过于宽泛，不仅 Emacs 为容纳这些帮助信息而创建的编辑缓冲区会过于巨大，整个操作的时间也会非常的长。比如说，如果给 **apropos** 命令的参数是“file”或“buffer”，就能让 Emacs “昏迷”一段时间（这要看计算机的速度有多快了），而且罗列在输出内容里的 Emacs 数据对象可能会有 100 多个。如果用 **super-apropos** 命令查找关于“buffer”的资料，它的输出将超过 6000 行！一般来说，可能需要调用好几次 **apropos** 命令才能找到真正想要的东西。

需要用到 **apropos** 和 **super-apropos** 命令的场合并不多，除非你是一位 Emacs LISP 程序员并且确实需要查找某些非命令函数的资料。但是，因为 Emacs 并没有提供



variable-apropos 之类的命令，所以如果想查找关于变量的资料，就肯定要使用这两个命令。我们认为 Emacs 应该有一个这样的命令，所以我们自己动手写了一个 LISP 函数，下面就是它的代码：这段代码是从 **command-apropos** 命令的代码里直接节选出来的。我们还给出了把这个命令绑定到“C-h C-v”组合键上的 LISP 代码。如果你喜欢，可以把这段代码照抄到你的 “.emacs” 文件里；这样，你就可以在以后的 Emacs 工作中使用这个命令了：

```
(defun variable-apropos (string)
  "Like apropos but lists only symbols that are names of user-modifiable
variables. Argument REGEXP is a regular expression
that is matched against user variable names. Returns list of symbols and
documentation found."
  (interactive 'sVariable apropos (regexp))
  (let ((message
         (let ((standard-output (get-buffer-create "*Help*")))
           (print-help-return-message 'identity)))
        (if (apropos string t 'user-variable-p)
            (and message (message message)))))

(define-key help-map '\C-v' 'variable-apropos)
```

作为这个命令的一个例子，请按下“C-h C-v”组合键，然后在提示符处输入“auto-save”；Emacs 将给出关于 **auto-save-default**、**auto-save-interval**、**auto-save-timeout**、**auto-save-visited-file-name** 和 **delete-auto-save-files** 等变量的资料。如果想查看其中某个变量的可取值和完整的说明，请使用“C-h v”命令。

一般性资料

剩下的帮助命令负责提供关于 Emacs 的一般资料。我们把这些命令汇总在表 16-3 里。

表 16-3：用来查找一般性资料的帮助命令

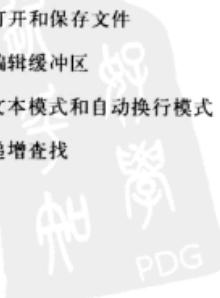
键盘操作	命令名称	动作
C-h t	help-with-tutorial	运行 Emacs 教程
C-h i	info	启动文档阅读器 Info 程序
C-h C-f	Info-goto-emacs-command-node	启动文档阅读器 Info 程序，并前进到指定的结点；这个帮助命令的参数是一个命令名

表 16-3：用来查找一般性资料的帮助命令（续）

键盘操作	命令名称	动作
C-h C-k	Info-goto-emacs-key-command-node	启动文档阅读器 Info 程序，并前进指定的结点；这个帮助命令的参数是对应于某个命令的按键组合
C-h n	view-emacs-news	查看关于 Emacs 新增和改进功能的信息
C-h F	view-emacs-FAQ	查看 Emacs 的常见问题答疑文件
C-h p	finder-by-keyword	沿着这个命令弹出的菜单可以查到关于安装在本系统上的 Emacs LISP 程序包的资料
C-h C-c	describe-copying	阅读 Emacs 的“通用公共许可证”(General Public License，参见附录六)
C-h C-d	describe-distribution	阅读从自由软件基金会订购 Emacs 的信息（参见附录一）
C-h C-p	describe-project	查阅关于 GNU 软件开发项目的信息
C-h C-w	describe-no-warranty	查阅 Emacs 的免责条款（参见附录六中的“NO WARRANTY”部分）

对初学者来说，这些命令中最重要的是“C-h t”（命令名是 **help-with-tutorial**），它会把屏幕上多余的窗口都删除（只留下一个）并开始教授使用 Emacs。事实上，它会在窗口里显示一个名为“*TUTORIAL*”的文件。Emacs 初学者可以通过这个教程熟悉以下一些 Emacs 功能的使用方法：

- 基本的光标移动
- 删除和恢复
- 打开和保存文件
- 编辑缓冲区
- 文本模式和自动换行模式
- 递增查找



- 基本的帮助命令

读者可以把这个教程与本书的第一章和第二章结合起来学习。这个教程很有帮助作用，虽然它只介绍了最基本的操作，但却是非常必要的。

如果想查阅更有深度的一般性文档，请按下“**C-h i**”组合键（命令名是 **info**），这个命令将启动文档阅读器 **Info** 程序。这个浏览器有点类似于现代的超文本系统（注 4），可以用来查看 Emacs（以及其他 GNU 软件）的文档资料。它是由信息结点（node）组成的一棵信息“树（tree）”。如果想查阅关于某个问题的资料，就需要选中与之对应的信息树；这棵信息树的结点则包含着这个问题的分论点、分分论点……以此类推；整个文档形成一种树状的结构。

按下“**C-h i**”组合键之后，Emacs 将在一个 **Info** 模式下的窗口里创建一个只读编辑缓冲区，里面是 **Info** 系统的目录结点（directory node）。虽然有关这个系统详细的使用方法超出了本书的讨论范围，但 **Info** 系统的自备文档很充足。如果在 **Info** 系统里按下“**h**”键，就会进入 **Info** 系统的教程。这个教程与我们刚才介绍的 Emacs 基本命令教程很相似。在这个编辑缓冲区里按下“**C-h m**”组合键（命令名是 **describe-mode**）就能获得关于 **Info** 模式的帮助信息；它对 **Info** 模式下的各种命令进行了汇总和介绍。

Emacs 帮助命令“**C-h C-f**”（命令名是 **Info-goto-emacs-command-node**）和“**C-h C-k**”（命令名是 **Info-goto-emacs-key-command-node**）能够进行更有针对性地使用。可以把它们看做“**C-h f**”（命令名是 **describe-function**）和“**C-h k**”（命令名是 **describe-key**）在 **Info** 系统里的等效命令；它们的作用是启动 **Info** 系统并根据给出的参数直接到达 **Info** 系统的某个结点，“**C-h C-f**”查阅用其参数指定的命令，“**C-h C-k**”查阅绑定到指定按键组合上的命令。

对忠实的 Emacs 用户和定制者来说，剩下的 Emacs 帮助命令中最重要的大概就是“**C-h n**”（命令名是 **view-emacs-news**）了，它将打开 Emacs 自带的文件 *NEWS* 以供查阅。这个文件其实是一个修改记录，自上一个主版本（major version）推出之后又在 Emacs 上做的改进都会记载在里面，比如 Version 18 到 Version 19 之间的全部修改以及 Version 19 到最新的次版本（minor version，比如笔者使用的 Version

注 4：事实上，它是具有实用价值的超文本系统中最早出现的成功范例之一。



19.30) 之间的全部修改。如果自上一个主版本推出之后曾经有过好几个次版本，这个文件的长度就会相当可观——笔者手里的这个文件有2500多行。如果想从这个文件里找出某个具体的 Emacs 修改事项，那么还是使用一个适当的查找命令比较好。如果只是随便看看，请注意它是按大纲模式的要求缩进的——主论点全都以“*”开头，分论点全都以“**”开头，以此类推。因此，最好是用大纲模式来浏览，具体做法请参考第八章中的有关内容。

针对复杂 Emacs 命令的帮助功能

许多比较复杂的 Emacs 命令都自备有一套专用的键盘命令。而这些命令又几乎都自带它们自己的帮助功能，但用来启动这类帮助功能的键盘命令是问号 (?) 而不是标准的 Emacs 帮助键。我们把 “?” 在几个常用的复杂命令中的作用汇总在下面：

dired (“C-x d” 组合键)

辅助输入区里将列出一些最常用的命令。注意：这个清单并不完整。按下 “C-h m” 组合键（命令名是 **describe-mode**）可以看到更全面细致的文档。

query-replace (“ESC %” 组合键）

它的子命令清单将显示在一个 “*Help*” 窗口里。按下 “C-h” 组合键（不管是否重新绑定过帮助键）也有同样的效果。这种做法也适用于 **query-replace-regexp** 命令。

save-some-buffers (“C-x s” 组合键）

与刚介绍的 **query-replace** 命令中的情况差不多。

list-buffers (“C-x C-b” 组合键）

你会在一个 “*Help*” 窗口里看到关于编辑缓冲区菜单模式的帮助信息。这个命令与按下 “C-h m” 组合键（命令名是 **describe-mode**）的效果相同。

自动补足功能

根据 Emacs 给出的提示在辅助输入区里输入事物的名称时，如果 Emacs 能够对它进行自动补足，就可以随时输入 “?”。这将打开一个 “*Completions*” 窗口，里面是当时可用的各种候选项。自动补足功能我们已经在本章的开头部分介绍过了。

附录一

如何获得 Emacs 软件

正如我们在本书前言里所说的那样，Emacs 是一个自由软件。这并不意味着可以完全免费地得到它，但自由软件基金会（Free Software Foundation, FSF）努力让大家以最可能容易的方式获得 Emacs。如果从 FSF 那里获得 Emacs，那么它会收取一定的费用——但收费的数额只是同类软件价格的一个零头。其目的是为了支付存储介质、运输和手续方面的成本。如果从朋友那里获得一份复本更方便，FSF 是鼓励你这样做的。如果你的周围环境里有很多机器（比如一家大公司或者大学里的某个系）并且已经有人拥有了 Emacs，从他或她手里获得 Emacs 肯定更受欢迎。

Emacs 能够运行在超过 70 种的 UNIX 版本上；它还已经被移植到 VAX/VMS、NextStep、Macintosh、Microsoft Windows 95 和 NT 等系统上，甚至被移植到 MS-DOS 上。如果你必须从本附录所介绍的这几个来源获得 Emacs，那么，从源代码（即用程序设计语言 C 和 LISP 写出来的程序）开始来建立它的工作可能就需要由你本人来完成。别紧张！这工作并不要求你非得是一名程序员。问题的关键是 Emacs 源代码的规模非常大，而你必须根据自己操作系统和硬件的组合情况对它们进行正确的配置。这个过程很消耗时间，但就大多数情况来看都不复杂。

用 FTP 从因特网上下载

如果你周围没有方便的 Emacs 来源而你又不想增加开支，那么最简单的办法就是从因特网上下载。如果你能够上网，就可以从众多站点中的某一个站点获得

Emacs —— 不过，你的网络连接必须快得足以传输一个 10 MB 字节的文件才行。你的硬盘至少要有 105 MB 剩余空间；今后发行的 Emacs 的规模肯定会超过这个数字。因特网上有许多站点都主动向外提供 Emacs 和其他一些 FSF 软件的源代码。

从某个因特网站点下载 Emacs 需要使用 FTP。需要特别说明的是，应该使用 FTP；支持匿名 FTP 的站点允许任何人连接，不过可能会有一些（可以理解的）访问限制规定。FTP 有 3 种使用方式，可以任选一种：

- 有一个 WWW 浏览器，比如 Mosaic 或 Netscape Navigator 等，它们能够支持 FTP 并提供图形化的用户操作界面。
- 在系统上实现因特网 TCP/IP 协议的软件自备一个图形化的 FTP 用户操作界面。
- 作为最低要求，至少应该能够从 UNIX 操作系统的 shell 命令提示符处访问 FTP。

有许多不同的站点都能够提供 Emacs 的源代码，但最正宗的站点是位于麻省理工学院（MIT）的 *prep.ai.mit.edu*，文件路径是 */pub/gnu* 子目录。你应该先尝试从这个站点进行下载，尤其是当你身处美国的时候。如果你不在美国，或者从这个站点进行下载时遇到了问题，世界各地还有很多 *prep.ai.mit.edu* 的镜像站点。

自由软件基金会有一份所有镜像站点的名单。如果你想获得这份名单以及与获取 Emacs 和其他 FSF 软件有关的其他有用信息，请输入命令 “**finger fsf@prep.ai.mit.edu > fsf-info**”。你将得到一个名为 *fsf-info* 的文件，里面包含着全部信息（当然，可以使用一个不同的文件名）。目前，这份名单开列了二十几个 FTP 站点，它们分布在加拿大、欧洲、南美洲、南部非洲和环太平洋地区。

不管访问的是哪一个站点，都必须在一个给定的目录里找到一个名为 *emacs-19.N.tar.gz* 的文件，其中的 *N* 是一个数字。这是 Emacs 发行版本的压缩档案；*N* 是次版本号。如果这个名字的文件不止一个，那么请下载版本号最高的文件。这个文件的长度大概是 10 MB。

下面是一个使用 WWW 浏览器下载 Emacs 的例子。我们假设使用 Netscape Navigator 从站点 *prep.ai.mit.edu* 处进行下载。

如果 Netscape Navigator 还没有启动，请启动之。点击 “**Location (地址)**” 栏并输



入 “<ftp://prep.ai.mit.edu/pub/gnu>”。这是该FTP站点的URL地址（Universal Resource Locator，统一资源定位器）；FTP站点的URL地址格式是“<ftp://site.name/directory>”（注1）。按回车键让Netscape链接到这个URL地址。稍等一会儿，就应该链接到prep.ai.mit.edu站点上。

在Netscape完成主页加载工作之后，你将看到一个文件清单。根据我们刚才给出的文件名在清单里找到那个文件。在文件名上点击鼠标键，屏幕上弹出一个对话框，询问是立即运行或保存这个文件、取消操作、还是配置一个“查看器”以运行这个文件。选择保存，然后选择用来保存这个文件的目录。点击“OK（确认）”，Netscape将开始进行文件传输。

如果你没有Web浏览器或其他图形化界面的FTP工具，就只能从UNIX操作系统的shell命令行来执行FTP操作。下面就是一次完成这项工作的UNIX会话过程；你输入的命令用粗体表示，系统响应则用普通字体表示。此外，我们将假设你的UNIX提示符是“\$”。

注意：作为一个良好的因特网公民，你应该在口令字提示符处输入你真实的电子邮件地址；不要直接按回车键。

```
$ ftp prep.ai.mit.edu
Connected to prep.ai.mit.edu.
220 aeneas FTP server (Version 4.136 Mon Oct 31 23:18:38 EST 1988) ready.
Name (prep.ai.mit.edu): anonymous
331 Guest login ok, send ident as password.
Password: your full email address
(An introductory message appears here.)
ftp> binary
200 Type set to I.
ftp> cd /pub/gnu
250 CWD command successful.
ftp> ls emacs-19.[0-9][0-9].tar.gz
200 PORT command successful.
150 Opening data connection for /bin/ls (128.119.40.122,2653) <0 bytes>.
emacs-19.28.tar.gz
emacs-19.29.tar.gz
emacs-19.30.tar.gz
226 Transfer complete.
remote: emacs-19.[0-9][0-9].tar.gz
60 bytes received in 0.014 seconds (4.2 Kbytes/s)
```

注1： 在某些Web浏览器上，这个格式是“<ftp://site.name/directory>”。



在上面的文件名里，辅版本号（28、29和30）可能会有所不同，但我想下载的是最新的版本（本例中就是30）。

```
200 PORT command successful.  
150 Opening BINARY mode data connection for emacs-19.30.tar.gz (10039387 bytes).
```

当文件传输开始之后，根据你因特网链接的速度，可能需要等待一会儿；可能是几分钟（对T-1专线而言）或超过一个小时（对一台28.8K-bps的调制解调器而言）。

等到文件传输结束，你将看到下面的信息：

```
226 Transfer complete.  
local: emacs-19.30.tar.gz remote: emacs-19.30.tar.gz  
10039387 bytes received in 1.7e+02 seconds (59 Kbytes/s)  
ftp>
```

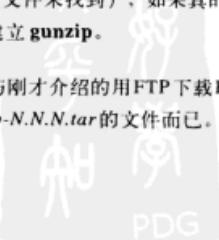
用“quit”命令退出FTP。如果使用的是Web浏览器或其他图形界面的工具，请根据自己的具体情况决定是否要退出程序。

解压缩和拆包

下载这个文件之后，真正开始建立Emacs之前还需要做两件事情：解压缩和拆包。这些工作的第一步需要使用一个名为**gunzip**的程序，但遗憾的是，它不是一个标准的UNIX命令。FSF软件的源代码文件都是用他们自己编写的一个名为**gzip**的程序来压缩的，没有使用UNIX操作系统中标准的**compress**工具。为什么会这样？这是因为标准的**compress**命令所使用的数据压缩算法是有专利的，不符合FSF倡导的自由软件精神（参见附录六）。另外一个原因是**gzip**的压缩效果和速度远远好于标准的**compress**命令。

gunzip是与**gzip**相对的解压缩工具。如果运气够好，你的系统上就应该已经有这个程序了。为了检查一下是否如此，请输入“**gunzip -h**”命令。如果看到**gunzip**的帮助信息和它的各种选项，就说明你系统上已经有这个程序了。否则，你将看到一条出错信息“not found（文件未找到）”；如果真的收到这条出错信息，就必须在做其他事情之前先下载和建立**gunzip**。

gunzip程序的下载过程与刚才介绍的用FTP下载Emacs的过程完全相同，只不过这次要找的是一个名为**gzip-N.N.N.tar**的文件而已。这个文件同时包含**gzip**和**gunzip**



两个程序的源代码。如果必须以命令行方式来完成 FTP 操作，则可以用命令“**ls gzip***”帮忙找到正确的文件。这个文件的长度在 800 KB 左右。

下载 **gzip-N.N.N.tar** 文件之后，需要对它进行拆包和安装。这个过程与 Emacs 的建立过程很相似，这是因为 FSF 制定有一套标准的安装流程，所有 FSF 软件都可以用同一种方法建立。具体步骤请查阅本附录后半部分中的“建立 Emacs”一节。为了把 **gzip** 和 **gunzip** 正确地安装到系统上，你必须请一位系统管理员帮忙。

把 **gunzip** 程序安装到系统上之后，就可以使用它和 **tar** 命令对 Emacs 进行解压缩和拆包操作。进入存放下载的 Emacs 源代码的那个目录。然后输入以下命令，你将看到一个文件清单：

```
$ gunzip -c emacs-19.30.tar | tar xvf -
x emacs-19.30/GETTING.GNU.SOFTWARE, 4737 bytes, 10 tape blocks
x emacs-19.30/INSTALL, 24397 bytes, 48 tape blocks
x emacs-19.30/PROBLEMS, 57880 bytes, 114 tape blocks
x emacs-19.30/README, 3960 bytes, 8 tape blocks
x emacs-19.30/BUGS, 937 bytes, 2 tape blocks
x emacs-19.30/move-if-change, 129 bytes, 1 tape blocks
x emacs-19.30/ChangeLog, 109081 bytes, 214 tape blocks
x emacs-19.30/Makefile.in, 20393 bytes, 40 tape blocks
x emacs-19.30/configure, 108386 bytes, 212 tape blocks
x emacs-19.30/configure.in, 40490 bytes, 80 tape blocks
. . .
```

这个文件清单得在屏幕上走一阵子——Version 19.30 大约包含有 1500 个文件。如果不想要看到这个文件清单，就可以把上面那条 **tar** 命令里的字母 “v”（对应 **verbose** 选项）删除。等这个命令执行完毕的时候，获取 Emacs 源代码文件的工作就算告一段落。

自由软件基金会

如果你不能访问因特网，就可以直接从 FSF 购买 Emacs。它们销售的 Emacs 有好几种存储格式，其中包括 CD-ROM 盘、软盘（用于 MS-DOS 版本）以及以下几种数据磁带，它们全都用 **tar** 命令进行了打包：

- 1600bpi 或 6250bpi 的 1/2 英寸双轴数据带
- Sun 公司出品的 DC300XLP 型 1/4 英寸盒式数据带（QIC 24）

- 惠普公司出品的 16 轨 DC600HC 型 1/4 英寸盒式数据带
- IBM 公司出品的 RS/6000 型 1/4 英寸盒式数据带 (QIC 150)
- Exabyte 公司出品的 8 毫米盒式数据带
- DAT 公司出品的 4 毫米盒式数据带

FSF能够根据顾客要求提供多种产品和服务组合，订购服务也算其中的一项。他们通常只提供源代码 (DOS软盘版是个例外，它包含一个已经建立好的可执行程序)；但如果用户报出自己的软、硬件配置情况，他们也愿意根据定单替用户建立Emacs。这些产品的价格（在本书写作期间）是：对个人用户，一张 CD-ROM 盘 \$60 美元；按定单制作的 Emacs 则收费可能高达 \$5000 美元。CD-ROM 盘上还有很多其他的 FSF 程序和文档，而数据磁带上除 Emacs 外就只有很少几个关键的程序设计工具和 **gzip** 软件。

FSF 的销售产品还包括《Gnu Emacs Reference Manual》(GNU Emacs 参考手册) 的副本，各种其他 GNU 软件和文档等等。

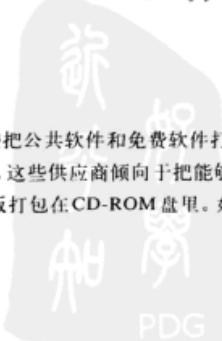
如果想进一步了解 FSF 或者订购他们的产品，请按下面的地址与他们联系：

Free Software Foundation, Inc.
59 Temple Place, Suite 330
Boston, MA 02111-1307
phone: (617) 542-5942
fax: (617) 542-2652
email: *gnu@prep.ai.mit.edu*

也可以访问 FSF 位于 <http://twws1.vub.ac.be/studs/tw45639/fsf.htm> 的 Web 主页。

其他 CD-ROM 来源

如果不考虑FSF渠道，你还可以选择一些把公共软件和免费软件打包到CD-ROM 盘里再以很合理的价格进行销售的小公司。这些供应商倾向于把能够在大多数比较流行的软、硬件平台上运行的Emacs预装版打包在CD-ROM 盘里。如果你有一个CD-



ROM 光驱，这类来源可能更值得考虑。如果你手里没有可以用来自行建立 Emacs 的工具（比如 C 编译器），就只能满足于一个预装版；即使你有必要的工具，这类来源也是一个方便的选择，它们能够节省大量的时间、硬盘空间并减少潜在的麻烦。下面是其他一些 Emacs 的 CD-ROM 发行版本来源：

Prime Time Freeware
370 Altair Way, Suite 150
Sunnyvale, CA 94086
phone: (408) 433-9662
fax: (408) 433-0727
email: *info@ptf.com*
WWW: <http://www.cfcl.com/ptf/>

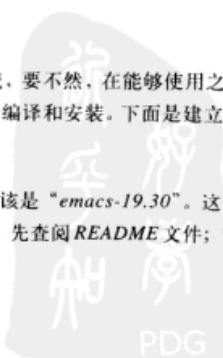
Ready-to-Run Software
4 Pleasant Street
Forge Village, MA 01886
phone: (800) 743-1723 or (508) 692-9922
fax: (508) 692-9990
email: *info@rtr.com*
WWW: <http://www.rtr.com/>

还可以从 O'Reilly & Associates 出版公司出版的《UNIX Power Tools》一书附赠的 CD-ROM 盘上获得 Emacs，这本书的作者是 Jerry Peek、Tim O'Reilly 和 Mike Loukides。

建立 Emacs

除非你的 Emacs 预装版正好适用于你的系统，要不然，在能够使用之前，你必须从源代码开始对 Emacs 众多的可执行组件进行编译和安装。下面是建立 Emacs 时应该注意的一些问题。

Emacs 源代码有一个顶级目录，它的名字应该是 “*emacs-19.30*”。这个目录里有两个名字分别是 *INSTALL* 和 *README* 的文件。先查阅 *README* 文件；它的内容通常



是很有用的一般性信息和“最后一分钟”才确定下来的重要“游戏规则”。接着查阅 *INSTALL* 文件，它为 Emacs 每个建立步骤给出了详细的指导；即使你不是一位 UNIX 专家，也应该看得懂。

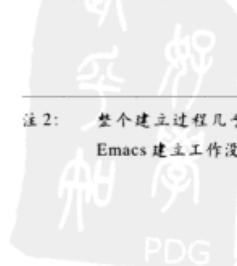
Emacs 的建立过程要花相当长的时间；大概得两三个小时。希望大家在整个过程中一切顺利。但各种意外情况还是有可能发生，比如编译器报告有个错误并半途而废、原以为有的某个系统文件无法找到、配置文件和系统的具体情况差之毫厘、或者是其他一些不可预见的陷阱等（注 2）。如果遇到麻烦，最好找一位身边的专家来帮助你。我们这本书不可能把安装过程中可能遇到的各种问题都面面俱到地讨论到。

FSF 的标准安装流程一直在改进，它越来越能干，对意外情况的处理也越来越好。但 Emacs 建立工作的难易程度主要取决于用户的软、硬件组合情况。FSF 的安装脚本里有一个名为 **configure** 的程序，它的作用是检查用户系统，确定用户所使用的硬件和软件，然后对 Emacs 做相应的配置。

configure 对系统配置情况的正确识别率是很高的。如果你的软、硬件组合属于一种比较主流的配置情况（比如 CPU 是 Sun 公司出品的 SPARC 芯片，操作系统是该公司最新推出的 Solaris UNIX 标准版本）并且你的系统管理员也没有对系统配置做过重大更改，这个正确率就更有保证。如果情况是这样，你的 Emacs 就应该能很顺利地建立起来，既不用做很多的配置设置，也不用技术专家来救急。可如果你有一个不同寻常的配置——比如一种很罕见的计算机或操作系统版本、一种不同寻常的软、硬件组合或者一种不常见的系统配置情况——那么除了你自己去配置这个软件之外就没有什么别的选择了。

我们的建议是：如果你了解 C 语言、是一位有经验的 UNIX 用户、并且愿意花时间去研究，那就放手大干一场吧！Emacs 是聪明能干而又高度复杂的软件中的一个典型例子。这么说吧：UNIX 操作环境能够提供的基础功能它几乎都用上了。UNIX 软件开发工具能够提供的开发手段它几乎也都用上了。钻研 Emacs 的源代码可能是你能够感受到的最高深的 UNIX 教育。

注 2：整个建立过程几乎总是会给出几条警告信息。你可以把它们看做坏消息，但如果 Emacs 建立工作没有因它们而半途而废，也不必太把它们放在心上。



附录二

解除他人对 Emacs 的定制设置

在学习 Emacs 的时候，最让人头疼的是：当你坐在终端前面时，才发现有位“热心肠”的系统管理员已经把所有的命令都改了。对这种情况你往往是无计可施的：你无法删除全局性的定制文件、你的 Emacs 与随机手册（或者本书）里讲的不同、而那位“热心肠”的系统管理员可能根本就没想过要把他做的修改记载下来。

如果给你开设账户的系统管理员已经为你准备了一个“有用的”个人 “*.emacs*” 配置文件，类似的问题就也有可能发生。这个“有用的”配置文件对有经验的用户来说可能是好东西，可如果因为有了它而反让你无法学用 Emacs 编辑器，那么它对你就没有什么帮助和价值。

还好，有一种办法能够让你从这些困境中解脱出来。如果你发现自己处于这种境地，先删除或者重命名你手里的所有 “*.emacs*” 文件。然后在你的主目录里创建一个只有一行语句的 “*.emacs*” 文件，这条语句必须严格按下面这样写：

```
(setq inhibit-default-init t); 不进行全局初始化
```

然后，重新启动 Emacs。这个文件的作用是制止 Emacs 读取它的全局初始化文件。

还有一种棘手的情况：如果你正坐在别人的键盘前面，又该怎么办？你启动了 Emacs，可出现在你面前的却都是别人“专用的”按键绑定和功能命令。这种困境也有解决办法：

- 试试“**emacs -q**”命令。“-q”选项的作用是告诉 Emacs 在启动之前不读取该用户的“*.emacs*”文件。这将使你避开该用户个人的定制设置。
- 但这个步骤并不会得到你自己的定制。如果想让 Emacs 读入你自己的“*.emacs*”文件，哪怕你使用的是别人的账户，请输入命令“**emacs -u yourname**”。比如，“**emacs -u deb**”将以用户 Deb 的初始化文件（即`~deb/.emacs`文件）启动 Emacs。

但是如果你是在一个网络上，则“-u”选项可能不起作用。它要求你在网络的每一个系统上都有同样的主目录，或者是在每一个系统上有不同的主目录但这些主目录里的“*.emacs*”文件都是最新的。这两个条件在某些网络上是无法满足的。况且，网络配置也不在本书的讨论范围之内。如果你对这个问题感兴趣，可以去看看 O'Reilly & Associates 出版公司出版的《Managing NFS and NIS》一书，它的作者是 Hai Stern；这本书对网络上的主目录问题做了详细的讨论。



附录三

Emacs 变量

我们已经在第十一章对 Emacs 变量进行过专门的讨论，本附录是对它的一个补充。我们将把与前十一章内容有关的 Emacs 变量根据常用、适用的原则汇总在以下的表格里。

表格中的变量是根据其用途分类的，它们的缺省值列在表格的第二栏里。具体到某个变量的详细说明请参考表格标题中给出的有关章节中的讨论。程序设计语言编辑模式所使用的变量的有关资料请查阅第十二章内容。

表 C-1：备份、自动保存和版本控制（参见第二、十五章）

变量	缺省值	说明
make-backup-files	t	如果取值为“t”，则在当前文件第一次存盘之前先给它创建一个备份版本
backup-by-copying	nil	如果取值为“t”，则不采用把将要保存的文件重命名为备份版本的做法，而是以复制的方法来创建备份文件。缺省的重命名办法比较有效率，而复制的办法则更安全——复制操作过程中发生的磁盘故障不会有什么损害，但在 mv 操作中“及时”出现磁盘故障却可能彻底毁坏备份文件。 ²
version-control	nil	如果取值为“t”，则创建文件的编号版本作为备份（备份的文件名格式是 “filename~N~”）。如果取值为“nil”，则只对已经有编号版本的文件这样做。如果取值为“'never”（注意前导的单引号），就不创建任何编号版本

表 C-1：备份、自动保存和版本控制（参见第二、十五章）（续）

变量	缺省值	说明
kept-new-versions	2	Emacs 在创建一个新的编号备份时将为文件保留的最新版本的个数
kept-old-versions	2	Emacs 在创建一个新的编号备份时将为文件保留最老版本的个数
delete-old-versions	nil	如果取值为“t”，则删除多余备份版本（根据上面两个变量的设置情况而需要保留的不算在内）时不需要用户确认；如果取值为“nil”，则需要经过用户确认
auto-save-default	t	如果取值为“t”，则对每个被访问文件都进行自动保存
auto-save-visited-file-name	nil	如果取值为“t”，则自动保存为被访问文件而不是保存为另外一个特殊的自动保存文件
auto-save-interval	300	两次自动保存操作之间间隔的按键次数。如果取值为“0”，则关闭自动保存功能
auto-save-timeout	30	Emacs 在用户没有操作情况下执行自动保存操作的时间间隔。如果取值为“nil”或“0”，则关闭此项功能
delete-auto-save-files	t	非“nil”值表示在保存“真正的”文件时删除其自动保存文件
buffer-offer-save	nil	非“nil”值表示在退出 Emacs 时提示用户保存当前编辑缓冲区，不论在它里面打开的是不是一个文件
vc-default-back-end	RCS	vc 程序包使用的版本控制系统。它的可取值是符号“RCS”、“CVS”或“SCCS”
vc-display-status	t	如果取值不为“nil”，则在状态行上给出版本号和版本控制状态
vc-keep-workfiles	t	如果取值不为“nil”，向版本控制系统注册所做的修改后，Emacs 将不删除工作文件
vc-mistrust-permissions	nil	如果取值不为“nil”，则认为文件自身的属主 ID 和访问权限标志不代表版本控制系统对此的看法；VC 将直接从版本控制系统那里获取这些信息
vc-suppress-confirm	nil	如果取值不为“nil”，则执行版本控制操作前不要求用户确认
vc-initial-comment	nil	如果取值不为“nil”，则在文件第一次向版本控制系统注册时提示输入一条初始注释

表 C-1：备份、自动保存和版本控制（参见第二、十五章）（续）

变量	缺省值	说明
vc-make-backup-files	nil	如果取值不为“nil”，则仍为已经用版本控制操作注册过的文件创建标准的 Emacs 备份版本
vc-consult-headers	nil	如果取值不为“nil”，则根据嵌在工作文件里的版本控制信息来确定版本号；否则从主控文件里获取此项信息

- a. 因为文件备份操作的效果是创建一个新的版本，所以新文件的属主就是你。换句话说，不管原始文件的属主是不是你，只要这个变量的取值是“nil”，以重命名方式得到的备份版本的属主也将是你。因此，如果你是以根用户身份进行的登录而你所编辑的文件的属主却不是根用户——比如 **uucp** 的配置文件，就可能会引起一些问题。

表 C-2：查找与替换（参见第三章）

变量	缺省值	说明
case-fold-search	t	如果取值不为“nil”，则在查找操作中不区分字母的大小写
case-replace	t	如果取值不为“nil”，则在替换时维持原来的大小写情况（对 case-fold-search 也不例外）
search-upper-case	'not-yanks	如果取值不为“nil”，则查找字符串中的大写字母将压制 case-fold-search 变量做出的设置（即强制查找操作区分字母的大小写）。符号“'not-yanks”的含义是把替换字符串中的大写字母转换为小写
search-exit-option	t	如果取值不为“nil”，则任何一个不是递增查找子命令 (DEL 、 LINEFEED 、 C-q 、 C-r 、 C-s 、 C-w 、 C-y) 的控制字符都将退出查找
search-slow-speed	1200	如果终端的通信速度等于或小于这个值，就将使用慢速递增查找——用一个小窗口来显示查找操作的部分结果 ^a
search-slow-window-lines	1	慢速查找所用窗口的高度，计量单位是文本行。如果这个数字是一个负数，就表示要把这个窗口放在屏幕的顶部而不是底部
search-highlight	nil	如果取值不为“nil”，则反显已经查找到的部分匹配。适用于 X 窗口系统和其他具备反显功能的显示器

表 C-2: 查找与替换 (参见第三章) (续)

变量	缺省值	说明
query-replace-highlight	nil	如果取值不为“nil”，则在查询-替换模式中反显找到的匹配。适用于X窗口系统和其他具备反显功能的显示器

- a. 遗憾的是，只有当计算机认为终端的通信速度低时，这个选项才能发挥作用。如果你的终端是通过LAN、终端服务器或其他数据通信设备进行通信的，就可能出现因计算机正在与“你的终端”进行高速通信而“忽略”“你的终端”速度慢的情况。

表 C-3: 屏幕显示 (参见第二、四、六章)

变量	缺省值	说明
next-screen-context-lines	2	用“C-v”或“ESC v”前、后卷屏时，Emacs 保留的上、下文行数
scroll-step	0	当光标沿垂直方向移出当前窗口时，Emacs 将前卷或后卷的文本行数。如果取值为“0”，则卷动足够的行以便光标在卷动后出现在窗口中央
hscroll-step	0	当光标沿水平方向移出当前窗口时，Emacs 将左卷或右卷的文本列数。如果取值为“0”，则卷动足够的列以便光标在卷动后出现在窗口中央
tab-width	8	制表位的宽度；如果被设置，将只对当前编辑缓冲区起作用
truncate-lines	nil	如果取值不为“nil”，则不对超长文本行进行自动换行，截断它并且用“\$”表示这一行超出屏幕画面
truncate-partial-width-windows	t	如果取值不为“nil”，则把显示宽度小于显示器宽度的全部窗口里的超长文本行截断（像上面那样）
window-min-height	4	窗口的最小高度（计量单位是文本行）
window-min-width	10	垂直分割窗口时，分割出来的窗口的最小宽度（计量单位是列）
split-window-keep-point	t	分割窗口时，非“nil”取值表示使两个窗口里的光标同步移动。如果取值为“nil”，则以使重新绘制屏幕画面的工作量最小为原则设置一个光标位置（这是为低速显示器准备的）

表 C-3: 屏幕显示 (参见第二、四、六章) (续)

变量	缺省值	说明
resize-minibuffer-mode	nil	如果取值不为“ nil ”，则允许辅助输入区增加高度以显示其中的内容
resize-minibuffer-window-exactly	t	允许辅助输入区的高度做动态改变，使之刚好能够显示其中的全部内容
resize-minibuffer-frame	nil	如果取值不为“ nil ”，则允许(X窗口系统显示器中的)辅助输入区窗格改变高度
resize-minibuffer-frame-exactly	t	允许(X窗口系统显示器中的)辅助输入区窗格的高度作动态改变，使之刚好能够显示其中的全部内容
resize-minibuffer-window-max-height	nil	resize-minibuffer-mode 模式中辅助输入区所能到达的最大高度；在X窗口系统下，如果这个变量的取值小于1或不是数字，则辅助输入区的高度不能超过它所在窗格的高度
ctl-arrow	t	非“ nil ”取值表示把控制字符显示为“^X”形式，其中的“X”是“被控制”的字符。其他取值表示把控制字符显示为八进制数字，比如说，“C-h”将被显示为八进制数字“\010”
display-time-day-and-date	nil	如果取值不为“ nil ”，则“ ESC-x display-time RETURN ”命令把星期几和日期都显示出来
line-number-mode	t	如果取值不为“ nil ”，则把行号显示在状态行上 ^a
line-number-display-limit	1,000,000	如果让 Emacs 显示行号，则编辑缓冲区的长度不能超过这个数字（计量单位是字符）
column-number-mode	nil	如果取值不为“ nil ”，则把列号显示在状态行上 ^a
visible-bell	nil	如果取值不为“ nil ”，则在必要时以屏幕闪烁代替蜂鸣报警
track-eol	nil	如果取值不为“ nil ”，则当光标位于某文本行的行尾并做上、下移动时，它仍将移动到前、后文本行的行尾；其他取值表示光标将固定在它此时所在的列位置
blink-matching-paren	t	如果取值不为“ nil ”，则用户输入一个需要配对出现的右括号类字符时，Emacs 将快速“闪现”与之配对的左括号类字符

表 C-3: 屏幕显示 (参见第二、四、六章) (续)

变量	缺省值	说明
blink-matching-paren-distance	4000	当用户输入一个需要配对出现的右括号类字符时, 向回查找其配对左括号类字符的最大距离 (计量单位是字符)
blink-matching-delay	1	闪现配对左括号类字符的持续时间
echo-keystrokes	1	如果用户输入命令时停顿了这个变量所设置的时间 (计量单位是秒), 则在辅助输入区里显示未完成命令的前缀 (比如 "ESC-") 作为提示; 取值为 "0" 时表示不提示
insert-default-directory	t	如果取值不为 "nil", 则在要求用户输入文件名时先把当前目录的路径名插入到辅助输入区里
inverse-video	nil	如果取值不为 "nil", 则对整个显示画面进行反显 (状态行将呈正常显示)
mode-line-inverse-video	t	非 "nil" 取值表示状态行将被反显
highlight-nonselected-windows	t	如果取值不为 "nil", 则把除当前窗口以外的所有窗口里的文本块反显; 适用于 X 窗口系统和其他具备反显功能的显示器
mouse-scroll-delay	0.25	当用户在某个窗口里按下鼠标并拖动到这个窗口边界以外的地方时, 屏显画面将延迟这个变量所设置的时间 (计量单位是秒) 之后才发生卷动。如果取值为 "0", 则表示以最快速度卷动
mouse-scroll-min-lines	1	当鼠标在窗口边界以外的地方被按下并做上、下拖动时, 至少要卷动这个变量所设置的文本行数

a. 可以用命令 "ESC-x line-number-mode" 和 "ESC-x column-number-mode" 切换行号、列号显示功能的开关状态。

表 C-4: 编辑模式 (参见第二、五、六、九、十一章)

变量	缺省值	说明
major-mode	fundamental-mode	新编辑缓冲区的缺省编辑模式。除非根据文件名后缀而另有规定; 设置这个变量的时候要注意在编辑模式名称的前面加上一个单引号 (这个变量的值必须是一个符号)
auto-mode-alist	(参见第十一章)	设置文件名和主编辑模式之间关联关系的列表
left-margin	0	在基本模式和文本模式里按下“C-j”组合键时的缩进量
indent-tabs-mode	t	如果取值不为“nil”, 则用户使用“C-j”组合键对文本进行缩进时允许使用制表符(以及空格)
find-file-run-dired	t	如果取值不为“nil”, 则在访问文件时, 如果用户文件名是一个目录, 就运行 dired
dired-kept-versions	2	在 Dired 里清理目录时, 需要保留的文件版本个数
dired-listing-switches	"-al"	生成 dired 文件清单时传递给 ls 命令的选项; 至少要包含 “-l” 选项
shell-file-name	\$SHELL	shell 的文件名; 如果某个 Emacs 函数需要调用 shell —— 比如 list-directory 、 dired 和 compile 等, Emacs 就会运行这个变量指定的 shell。此缺省值表示将使用 UNIX 环境变量 SHELL 的值
load-path		这个列表变量设置的是加载 LISP 程序包(参见第十三章)时的搜索路径; 通常就是 Emacs 源代码在系统的安装目录下的 <i>lisp</i> 子目录

表 C-5: 电子邮件 (参见第六章)

变量	缺省值	说明
mail-self-blind	nil	如果取值不为“nil”, 则自动把你的名字加入到“BCC (密抄)”栏以保证能给自己留下一份邮件副本

表 C-5：电子邮件（参见第六章）（续）

变量	缺省值	说明
rmail-mail-new-frame	nil	如果取值不为“nil”，则创建一个新窗格来编写外发邮件消息，仅适用于 X 窗口系统
mail-default-reply-to	nil	缺省插入到邮件消息“Reply-to:(回复地址)”栏里的字符串
mail-use-rfc822	nil	如果取值不为“nil”，则使用完全符合 RFC 822 标准规定的地址解析器对邮件地址进行解析；这会多花上一点时间，但正确解析出复杂网络地址的概率会有所增加
mail-host-address	nil	你的机器的名字；将被用来构造 user-mail-address
user-mail-address	(你的邮件地址)	你的完整的电子邮件地址
rmail-primary-inbox-list	nil	保存新收（尚未阅读）邮件的文件清单。如果取值为“nil”，则使用环境变量 \$MAIL 的值；如果 \$MAIL 指定的路径不存在，就使用路径 “/usr/spool/mail/yourname”
rmail-file-name	“~/RMAIL”	RMAIL 用来保存邮件消息的文件
mail-archive-file-name	nil	用来保存所有外发邮件消息的文件名字符串；如果取值为“nil”，表示不保存外发邮件
mail-personal-alias-file	“~/.mailrc”	用来保存邮件假名的文件名； Emacs 的邮件编辑模式与 UNIX 操作系统标准的 mail 和 mailx 程序使用的假名格式是一样的
mail-signature	nil	准备添加到外发邮件消息末尾的文本
rmail-dont-reply-to-names	nil	与这个正则表达式相匹配的名字将被排除在邮件回复地址名单之外；如果取值为“nil”，则把你本人排除在回复名单之外
rmail-displayed-headers	nil	与这个正则表达式相匹配的邮件消息标题栏将被显示出来；如果取值为“nil”，则把 rmail-ignored-headers 变量没有包括的标题栏都显示出来
rmail-ignored-headers	(复杂正则表达式)	不显示与这个正则表达式相匹配的邮件消息标题栏

表 C-5: 电子邮件 (参见第六章) (续)

变量	缺省值	说明
rmail-highlighted-headers	<code>"^From: ^Subject:"</code>	反显与这个正则表达式相匹配的邮件消息标题栏; 适用于 X 窗口系统和其他具备反显功能的显示器
rmail-delete-after-output	<code>nil</code>	如果取值不为 <code>"nil"</code> , 则自动删除已经被保存到某个文件里的邮件消息
mail-from-style	<code>'angles</code>	Emacs 为 <code>"From:(发信人)"</code> 栏生成的用户名的格式。如果取值为 <code>"nil"</code> , 则只包括电子邮件地址; 如果取值为 <code>"angles"</code> , 则把电子邮件地址用角括号括起来 (例如 <code>"Dave Roberts<d roberts@ed.com>"</code>); 如果取值为 <code>"parens"</code> , 则把电子邮件地址用圆括号括起来 (例如 <code>"Dave Robert(d roberts@ed.com)"</code>)

表 C-6: 文本编辑 (参见第二、三、六、九、十一章)

变量	缺省值	说明
sentence-end	(见第十三章)	匹配句尾的正则表达式
sentence-end-double-space	<code>t</code>	如果取值不为 <code>"nil"</code> , 则不把句号后面的单个空格看做句尾
paragraph-separate	<code>"[\t\C-\l]"</code>	匹配段落分隔行首的正则表达式
paragraph-start	<code>"[\t\n \C-\l]"</code>	匹配段落分隔行或段落第一行行首的正则表达式
page-delimiter	<code>"\C-\l"</code>	匹配分页符的正则表达式
tex-default-mode	<code>'plain-tex-mode</code>	打开 / 创建 TeX 或 LATEX 文件时将被启动的编辑模式
tex-run-command	<code>"tex"</code>	在 TeX 模式下, 用来运行 TeX 程序以排版某个文件的命令字符串; TeX 程序将运行在一个子进程里
latex-run-command	<code>"latex"</code>	用来运行 LATEX 程序的命令字符串; LATEX 程序将运行在一个子进程里
slitex-run-command	<code>"slitex"</code>	用来运行 SliTeX 程序的命令字符串; SliTeX 程序将运行在一个子进程里

表 C-6：文本编辑（参见第二、三、六、九、十一章）（续）

变量	缺省值	说明
tex-dvi-print-command	"lpr -d"	在 TeX 模式里，用 "C-c C-p" 组合键打印文件时将调用的命令字符串
tex-dvi-view-command	nil	用 "C-c C-v" 组合键查看 ".dvi" TeX 输出文件时将调用的命令字符串。在 X 窗口系统上，这个变量的值往往设置为 "xdvi"
tex-offer-save	t	如果取值不为 "nil"，则 Emacs 将在运行 TeX 之前提示用户保存尚未存盘的编辑缓冲区
tex-show-queue-command	"lpq"	在 TeX 模式里，用 "C-c C-q" 组合键查看打印队列时将被调用的命令字符串
tex-directory	".."	TeX 存放临时文件的目录；默认为当前目录
outline-regexp	"^\\[C-I]+"	在大纲模式里，用来匹配文本标题行的正则表达式
outline-heading-end-regexp	在大纲模式里，用来匹配文本标题行尾的正则表达式	
selective-display-ellipses	t	如果取值为 "t"，则把大纲模式里的隐藏文本显示为省略号 "..."; 其他取值表示什么也不显示

表 C-7：自动补足功能（参见第十六章）

变量	缺省值	说明
completion-auto-help	t	如果取值不为 "nil"，则在自动补足功能（辅助输入区里的 TAB 和 RETURN 键）无效或有二义时提供帮助
completion-ignored-extensions	(参见第十六章)	Emacs 将不对这个列表里的文件名后缀进行补足
completion-ignore-case	nil	如果取值不为 "nil"，则进行自动补足时将忽略字母的大小写

表 C-8: 杂项

变量	缺省值	说明
<code>kill-ring-max</code>	30	保存在删除环里的被删除文本块数；如果没有剩余的空间，就将删除其中最“老”的那个
<code>require-final-newline</code>	<code>nil</code>	如果某个已经被保存起来的文件没有最末尾的 LINEFEED 字符，则这个变量取值为 “ <code>nil</code> ” 时不自动添加之；取值为 “ <code>t</code> ” 时则自动添加之；其他取值时询问用户是否想添加一个 LINEFEED 字符 ^a
<code>next-line-add-newlines</code>	<code>t</code>	如果取值不为 “ <code>nil</code> ”，则在编辑缓冲区的末尾执行 next-line 命令（按下 “C-n” 组合键或向下方向键）时将插入一个新行而不是报告出错

- a. 注意，有些程序（比如 **troff**）要求文件必须以 **LINEFEED** 字符结束。



附录四

Emacs LISP 程序包

本附录对 Emacs 自带的最有用的 LISP 程序包进行了汇总。LISP 程序包通常都存放 在目录 “*emacs-source/lisp*” 里，而 “*emacs-source*” 代表存放 Emacs 源代码的 目录。我们略去了那些为 Emacs 提供“基本”功能支持的程序包；同理，我们也略去 了那些功能过时或者不易使用的程序包。

虽然书中已经详细介绍过几个程序包，但数量还是太少；读者必须依靠 GNU Emacs 的在线帮助功能查找对程序包功能的精确描述。我们在第十六章专门对 Emacs 的在 线帮助功能作了详细的介绍；具体到查找 LISP 程序包功能方面的资料，最重要的帮 助命令是 “**C-h p**”（命令名是 **finder-by-keyword**）、“**C-h f**”（命令名是 **describe- function**）和 “**C-h m**”（命令名是 **describe-mode**）。

“**C-h p**”的用处非常大。它可以帮助从相关的信息树上把所有能够用在自己系统上的 程序包的资料都查出来，其适用面从本附录所汇总的各种基本功能领域的程序包一直 延伸到每一种编辑模式的 “**C-h m**” 信息。因为篇幅的关系，我们在下面那些 表格对程序包做的“详细”说明还不够完整，而很多列在表格里的程序包却又可能 只有那些坚定的 Emacs 定制者才感兴趣。

只要合理，我们就会在表格里给出用来“启动”某个程序包的命令。表格中“启动 命令”一栏的含义是：

- 如果程序包是实现某个主编辑模式的，则它的启动命令就是把Emacs放到这个主编辑模式里的函数。
- 如果实现主编辑模式的程序包还会在访问带有某个特定后缀名的文件时被自动加载，我们就将在启动命令的后面再开列出“*suffix suffixname*”（普通字体的“*suffix*”中文意思是“后缀”；表格中的文件名后缀用斜体字表示）。
- 如果程序包是实现某个副编辑模式的，则它的启动命令就是把Emacs放到这个副编辑模式里的函数。
- 如果程序包实现了多个、成系列的通用性函数，我们将尽量把其中最“典型的”函数挑选出来。比如说，*studly* 程序包实现了 3 个命令；而我们将把 **studlify-region** 挑出来作为这个程序包的启动命令。如果没有更合理的选择，我们就写上“many”。

最后，我们还想说说程序包使用方面的问题。有些程序包是在 Emacs 启动时自动加载的，有些则需要等到你访问某个有特定文件名后缀的文件时才会加载（比如许多程序设计语言专用的编辑模式），有些是在你执行某个特定命令时自动加载的（例如“**ESC x shell RETURN**”加载对应于 **shell-mode** 的程序包 *shell.el*），还有一些则永远也不会自动加载。Emacs 对程序包的加载方式（即哪些程序包会在启动时自动加载，哪些会因用户操作而自动加载，哪些根本就不会被自动加载）事先有一个标准的配置，但你的系统管理员可能已经对这种配置进行了定制。如果是这样的话，你又该做些什么呢？其实，你不用关心这个问题，我们已经把程序包的启动命令列在下面这些表格里；你只要在启动 Emacs 之后发出这个命令（“**ESC x startup-command RETURN**”）试试就知道了。如果 Emacs 抱怨说“no match (找不到匹配)”，就说明这个程序包不能自动加载，必须由你来“亲手”加载它。如果你不想退出 Emacs，就需要使用命令“**ESC x load-library name RETURN**”，其中的“*name*”是我们在表格第一栏里给出的程序包名称；如果你想让 Emacs 在启动时自动加载这个程序包，就需要把下面这样一条语句添加到“*.emacs*”文件里：

```
(load-library "name")
```

好，不啰嗦了。研究这些表格里的 LISP 程序包去吧。

表 D-1：对 C 和 C++ 程序设计语言的支持

程序包名称	说明	启动命令
<i>cc-mode</i>	主编辑模式：编辑 C、C++ 和 Objective-C 源代码文件	c-mode, c++-mode, objc-mode, suffixes .c, .h, .y, .lex, .cc, .hh, .C, .H, .cpp, .cxx,
<i>c-fill</i>	副编辑模式：增加了对多行 C 程序注释的排版功能	c-comment
<i>cmacroexp</i>	函数：功能是用 cpp 扩展 C 源代码中的宏定义的函数	c-macro-expand
<i>hideif</i>	副编辑模式：隐藏 C 预处理器条件中的代码	hide-ifdef-mode
<i>cpp</i>	主编辑模式：反显和隐藏 C 预处理器条件中的代码；适用于 X 窗口系统和其他具备反显功能的显示器	cpp-parse-edit
<i>gud</i>	主编辑模式：与调试器 gdb 、 sdb 、 dbx 和 xdb 等配合工作	gud-mode

表 D-2：对其他程序设计语言的支持

程序包名称	说明	启动命令
<i>ada</i>	主编辑模式：编辑 Ada 源代码	ada-mode, suffixes .ada, .adb, .ads
<i>fortran</i>	主编辑模式：编辑 FORTRAN 源代码	fortran-mode, suffix .f
<i>asm-mode</i>	主编辑模式：编辑汇编源代码	asm-mode, suffix .s
<i>awk-mode</i>	主编辑模式：编辑 awk 源代码	awk-mode, suffix .awk
<i>f90</i>	主编辑模式：编辑符合 FORTRAN-90 标准的 FORTRAN 源代码	f90-mode, suffix f90
<i>fortran</i>	主编辑模式：编辑 FORTRAN 源代码	fortran-mode, suffixes .f, .for
<i>icon</i>	主编辑模式：编辑 ICON 源代码	icon-mode, suffix .icn
<i>mim</i>	主编辑模式：编辑 MIM 源代码	mim-mode
<i>modula-2</i>	主编辑模式：编辑 Modula-2 源代码	modula-2-mode

表 D-2：对其他程序设计语言的支持（续）

程序包名称	说明	启动命令
<i>perl-mode</i>	主编辑模式：编辑 Perl 源代码	perl-mode , suffix <i>.pl</i>
<i>prolog</i>	主编辑模式：编辑 Prolog 源代码	prolog-mode , suffix <i>.prolog</i>
<i>simula</i>	主编辑模式：编辑 Simula-87 源代码	simula-mode ,
<i>tcl-mode</i>	主编辑模式：编辑 tk/tcl 源代码	tcl-mode , suffix <i>.tcl</i>
<i>compile</i>	函数：编译程序和运行 make	compile

表 D-3：对 LISP 程序设计语言的支持

程序包名称	说明	启动命令
<i>lisp-mode</i>	主编辑模式：LISP、Emacs LISP 和 LISP 互动等模式	lisp-mode , emacs-lisp-mode , lisp-interaction-mode ^a
<i>scheme</i>	主编辑模式：编辑 Scheme 源代码	scheme-mode , suffixes <i>.scm</i> , <i>.oak</i>
<i>cl</i>	函数和宏：处理 Emacs LISP 和 Common LISP 之间的兼容问题	many
<i>debug</i>	主编辑模式：调试 Emacs LISP 程序	debug
<i>edebug</i>	实现为副编辑模式的 Emacs LISP 调试功能	edebug
<i>disass</i>	函数：对 Emacs LISP 编译代码进行反汇编	disassemble
<i>elp</i>	Emacs LISP 的代码检查器	elp-instrument-package , elp-instrument-function
<i>trace</i>	生成 Emacs LISP 程序的函数调用顺序表	trace-function
<i>cmuscheme</i>	面向 CMU Scheme 解释器的接口；CMU Scheme 解释器运行在一个子进程里	run-scheme
<i>xscheme</i>	面向 MIT Scheme 解释器的接口；MIT Scheme 解释器运行在一个子进程里	run-scheme

- a. LISP 模式在遇到后缀名为 “.l”、“.lsp”、“.lisp” 和 “.ml” 的文件时自动启动；Emacs LISP 模式在遇到后缀名为 “.el” 和 “.emacs” 以及任何被命名为 “_emacs”的文件时自动启动。LISP 互动模式是编辑缓冲区 “*scratch*” 的编辑模式。

表 D-4: 对文本处理的支持

程序包名称	说明	启动命令
<i>text-mode</i>	主编辑模式：编辑尚未排版的文本文件	text-mode^a
<i>nroff</i>	主编辑模式：编辑 <i>nroff</i> 和 <i>troff</i> 文本文件	nroff-mode
<i>scribe</i>	主编辑模式：编辑 Scribe 文本文件	scribe-mode
<i>sgml-mode</i>	主编辑模式：编辑 SGML 文本文件	<i>suffixes .sgm, .sgml, .dtd</i>
<i>tex-mode</i>	主编辑模式：编辑 TeX 和 LATEX 文本文件	tex-mode, latex-mode^b
<i>bibtex</i>	主编辑模式：编辑 LATEX 目录文件	bibtex-mode, suffix .bib
<i>refbib</i>	把 refer 格式的目录文件转换为 LATEX 格式	r2b-convert-buffer

- a. 文本模式在遇到后缀名为 “.text”、“.article” 和 “.letter” 以及带有前缀 “/tmp/Re”、“MessageN”（邮件消息）和 “/tmp/fol”（某些新闻阅读器会使用这个前缀）的文件时自动启动。
- b. TeX 模式在遇到后缀名为 “.tex” 和 “.TeX” 的文件时自动启动；LATEX 模式在遇到后缀名为 “.ltx”、“.sty”、“.cls” 和 “.bbi” 的文件时自动启动。

表 D-5: 对其他编辑器的仿真

程序包名称	说明	启动命令
<i>edt</i>	函数：设置仿真 VAX/VMS 平台上 EDT 编辑器的按键绑定	edt-emulation-on
<i>mlconvert</i>	函数：把 Gosling Emacs 的“伪 LISP”代码转换为 Emacs LISP 代码	convert-mocklisp-buffer
<i>vi</i>	主编辑模式：仿真 vi 编辑器	vi-mode
<i>vip</i>	另外一个用来仿真 vi 编辑器主编辑模式	vip-mode
<i>ws-mode</i>	主编辑模式：仿真 WordStar 文字处理器	wordstar-mode

表 D-6: 面向 UNIX 工具的接口

程序包名称	说明	启动命令
<i>shell</i>	主编辑模式：与 UNIX 操作系统的 shell 互动	shell-mode
<i>diff</i>	用 UNIX 的 diff 命令对比两个文件之间的差异	diff, diff-backup
<i>find-dired</i>	运行 UNIX 的 find 命令并在其结果文件清单上用 dired 做进一步的处理	find-dired

表 D-6: 面向 UNIX 工具的接口 (续)

程序包名称	说明	启动命令
<i>tar-mode</i>	通过一个 dired 风格的界面访问 tar 档案文件	tar-mode , suffix .tar
<i>lpr</i>	打印输出编辑缓冲区或文本块的内容	lpr-buffer , print-buffer , lpr-region , print-region
<i>uncompress</i>	函数: 访问压缩文件的过程中临时对文件进行解压缩	uncompress-while-visiting
<i>spell</i>	函数: 检查拼写情况	spell-word , spell-region , spell-buffer

表 D-7: 面向通信程序的接口

程序包名称	说明	启动命令
<i>ange-ftp</i>	函数: 通过 FTP 透明地访问远程文件	许多标准的文件操作命令
<i>kermit</i>	函数: 在 shell 模式里帮助运行数据通信程序 kermit	many
<i>mh-e</i>	与 MH 邮件系统配合使用的各种函数	mh-rmail , mh-smail
<i>rmail</i>	主编辑模式: 阅读和编辑电子邮件	rmail
<i>gnus</i>	主编辑模式: 阅读和投稿 Usenet 新闻	gnus
<i>telnet</i>	主编辑模式: 使用 Telnet 协议连接远程机器	telnet , rsh
<i>rlogin</i>	一个用来完成远程操作的 shell 模式的超集	rlogin

表 D-8: 通用编辑功能程序包

程序包名称	说明	启动命令
<i>chistory</i>	函数: 编辑和重复执行以前执行过的命令	repeat-matching-complex-command
<i>echistory</i>	主编辑模式: 编辑和重复执行以前执行过的命令	electric-command-history
<i>compare-w</i>	函数: 比较两个 Emacs 窗口	compare-windows
<i>options</i>	函数: 查看和设置 Emacs 变量	list-options , edit-options

表 D-8: 通用编辑功能程序包 (续)

程序包名称	说明	启动命令
<i>sort</i>	函数: 对各种文本文件进行排序	sort-lines , sort-columns , sort-fields
<i>outline</i>	主编辑模式: 编辑大纲	outline-mode
<i>underline</i>	函数: 给编辑缓冲区中的文本加下划线	underline-region

表 D-9: 游戏

程序包名称	说明	启动命令
<i>blackbox</i>	主编辑模式: Blackbox 游戏	blackbox-mode
<i>cookie</i>	函数: 根据月相算命	cookie
<i>dissociate</i>	函数: 随机拼凑各种文字	dissociated-press
<i>doctor</i>	主编辑模式: Psychiatrist (心理学家) 游戏	doctor
<i>dunnet</i>	主编辑模式: Adventure (冒险) 游戏。	dunnet
<i>flame</i>	函数: 随机出现的粗话	flame
<i>gomoku</i>	主编辑模式: Gomoku 游戏	gomoku
<i>hanoi</i>	函数: 汉尼塔游戏	hanoi
<i>life</i>	主编辑模式: Life 游戏	life
<i>mpuz</i>	主编辑模式: 随机拼图游戏	mpuz
<i>solitaire</i>	主编辑模式: 纸牌游戏	solitaire
<i>spook</i>	函数: 让 NSA (美国国家安全局) 注意你的邮件	spook
<i>studly</i>	函数: 把文章中的小写字母随机地改为大写字母	studify-region
<i>yow</i>	随机打印一个 Zippy the Pinhead (Emacs 中的游戏人物) 说的笑话	yow

表 D-10: 杂项

程序包名称	说明	启动命令
<i>calendar</i>	主编辑模式: 显示一个全功能的日历	calendar
<i>emacsbug</i>	函数: 向 FSF 报告发现一个程序漏洞	report-emacs-bug
<i>time</i>	函数: 在状态行上显示当前时间	display-time

附录五

软件漏洞及其修补

没有十全十美的程序。虽然 GNU Emacs 经受了非常彻底的调试，但肯定还能找出不能正确工作的环节。我们就曾在这本书的写作过程中设法让 Emacs “死”过一次——不过，和大多数用户一样，我们并没有把这个问题报告给有关方面：我们接着工作去了。

自由软件基金会对上报给他们的问题是重视的，反应也非常迅速。但他们需要的是真正的问题报告；针对某些具体做法的不同意见不能算做程序漏洞。如果你认为某个命令应该这样或者那样，那么请记住 Emacs 已经存在了相当长的时间而且已经有了大批的用户——为了迎合一两个人而改变它是不切实际的。为了便于用户反映程序漏洞，自由软件基金会在《GNU Emacs Manual》(GNU Emacs 手册)里给出了非常详细的指导，我们把它们总结如下：

- 如果你遇到一个系统级错误（比如 Emacs 死机、因内存分配问题而意外终止运行或者造成“反动”破坏），就表明 Emacs 很有可能存在一个程序漏洞。
- 在上报程序漏洞的时候，要把事情的原委尽可能地讲清楚。有些命令可以帮你把 Emacs 的故障现象描述得更精确。“C-h l”命令（命令名是 **view-lossage**）能够把最近输入的 100 个按键查出来；“ESC :”命令（命令名是 **open-dribble-file "filename"**）能够把按下的每一个按键组合存到指定的 “filename” 文件里，而“ESC :”命令（命令名是 **open-termscript "filename"**）能够把你输入的每一个键和发送到屏幕上的每一个字符保存到指定的 “filename” 文件里。

- FSF不鼓励由你本人对程序漏洞报告里的程序漏洞做出解释。你只要能讲清楚“我是这样、这样做的，然后发生了那样、那样的事情”就足够了，尤其是故障现象一再出现的时候；而“我认为终端处理存在这一问题”之类的说法却提供不了任何有用的信息。
- 在程序漏洞报告中，一定要清楚地说明你使用的Emacs的版本。可以用命令“**ESC x emacs-version**”查出有关的信息。
- 你还需要在程序漏洞报告里说明下面几件事：当时正在编辑的文件内容（如果有特殊之处）、“.emacs”文件的内容、当时所在的编辑模式以及错误发生之前刚加载的LISP程序包，等等。

我们再来加上一条重要的指导意见：

- 虽然我们已经尽了最大的努力来保证本书的准确性，但这并不代表它是十全十美的。我们自己也知道它距离完美还有一定的差距。因此，在上报程序漏洞的时候，千万不要以本书做为权威参照。虽然我们没有问过，但自由软件基金会极有可能会拒绝接受根据第三方出版物做出的程序漏洞报告。因此，如果你认为自己遇到了一个程序漏洞，请根据《GNU Emacs Manual》（GNU Emacs手册）或Emacs的在线帮助工具判断给你带来麻烦的命令的实际功用。你可能发现我们这本书讲错了；如果真是这样，请按电子邮件地址 *nuts@ora.com* 给我们发一份问题报告。

如果你能够通过因特网发送电子邮件，可以把程序漏洞报告发往：

bug-gnu--emacs@prep.ai.mit.edu

或者，你可以用程序包 **emacsbug** 提交程序漏洞报告。具体做法是：输入“**ESC x submit-bug-report**”命令，在提示符处输入报告主题，然后按回车键；屏幕上将弹出一个“Mail”窗口，其中的“Subject:(主题)”栏已经被填写为刚才在提示符处输入的内容。这是将邮件发往正确地址的简便方法。在Emacs里使用电子邮件的细节请参阅本书第六章内容。



附录六

Emacs 的版权文件

本附录将要介绍的两个重要文件可能大家都已经很熟悉了，它们就是《GNU通用公共许可证》(GNU General Public License) 和《GNU宣言》(GNU Manifesto)。《GNU通用公共许可证》对GNU Emacs、自由软件基金会的其他产品以及很多由个人编写和发行的程序的版权问题做出了规定；我们把这份许可证的全文收录在本附录里。《GNU宣言》是一份很长的文档，它对自由软件基金会的存在意义做出了声明。

在这些材料之后，我们还将对“自由编程联盟”(the League for Programming Freedom)组织做一个简单的介绍。这个组织与自由软件基金会并没有什么关系，但目标却差不多。我们将告诉大家从哪儿能获得这个联盟两份重要的时势分析报告，并把摘自其中一份报告的会员申请表收录在本附录里。

注意：有关文档完全按原文收录，一字未改。

《GNU 通用公共许可证》

GNU Emacs的一切复本都是在《GNU通用公共许可证》的规定下发行的，大家可能更熟悉“copyleft”(译注1)的叫法。根据《GNU通用公共许可证》的基本精神，

译注1：这是一个臆造出来的单词，与“版权”的英文单词“copyright”正好相对。

任何一位拥有 GNU Emacs 的人都有权向他人提供 Emacs 的复本；而任何一位收到 Emacs 的人都不得给 Emacs 发行版本增加任何限制规定；如果有人想发行任何针对 Emacs 的改进，就必须像原来的程序那样遵守同样的规定——即必须按《GNU 通用公共许可证》制度发行他或她所做的改进。为传播 Emacs 而收取费用是允许的；从这个方面看，“自由软件”并不一定就是廉价的。总之，你不能剥夺任何人（包括你的顾客在内）使用和传播这个程序的权利。这份许可证将保证 GNU Emacs 和其他程序永远都是自由的。

《GNU 通用公共许可证》已经成为 UNIX 文化的一个重要组成部分。它对计算机软件行业里一种常见的丑陋行径——找个不错的公共域软件，对它做些改进，然后把经过改进的程序视为一项私有财产——进行了抵制。程序员之所以会把他们辛辛苦苦写出来的程序放到公共领域里，其目的就是希望它们能够被更多的人所共享；可这种美好的愿望却因为办法的简单而给少数人以可乘之机。通过抵制上述行径，《GNU 通用公共许可证》把对软件作者这种意愿的保护向前推进了一大步。如今，希望与他人共享开发成果的程序员对软件进行“copyleft”（即采用《GNU 通用公共许可证》作为其发行许可证）的现象越来越普遍。

自由软件基金会的这份“通用公共许可证”在拟订时就考虑到要让任何程序都能很容易地使用它。在这份许可证正文的后面，自由软件基金会告诉你如何把它应用到你自己的软件上。FSF 会定期对这份许可证进行修订，而它目前正出于一种流动状态。我们这里提供的 Version 2 是它最新的版本。在这个版本之前，FSF 还曾推出过几个过渡性版本；此外，为解决“copylefting”函数库的许可证问题，FSF 还曾推出过一份特殊的版本。

《GNU 通用公共许可证》是一份非常精炼的法律文书。并且，正像这份许可证所承诺的那样，今后的版本仍将坚持下面这个版本里所体现的精神。它可能会在遣词造句等细节上有所变化，但你做为一名用户的基本权利肯定不会被动摇。

《GNU通用公共许可证》全文（译注2）

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software — to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

译注2：《GNU通用公共许可证》是一份法律文书，理应由自由软件基金会做出解释；为了避免不必要的误导和纠纷，我们没有翻译它。而如果读者想“copyleft”自己的软件，就应该把《GNU通用公共许可证》——当然是英文的，加入到它里面；这也是我们不翻译它的原因之一。

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution,

a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <19yy> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19yy name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items — whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit



linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

以上信息直接取自 Emacs 自带的 *COPYING* 文件，除了它的格式，我们没有对它的内容做任何修改。大家可以在 Emacs 发行版本的 *etc* 目录里找到这个文件。当然，根据 Emacs 发行版本的具体情况，某些细节可能会与这里有所不同。

《GNU 宣言》

自由软件基金会的另一份基本文件是《GNU 宣言》。这份宣言介绍了自由软件基金会存在的原因和目的。更可贵的是，它阐明了 FSF 对整个软件行业的看法。你可能不赞同它的观点，但你应该读读它。与刚才介绍的《GNU 通用公共许可证》相比，它的长度非同一般（虽然也不是太长），所以我们不准备把它的原文收录在这里。如果你在机器里保留了 GNU Emacs 的源文件，就可以在文件“*emacs-source*”/etc/GNU 里找到这份宣言，其中的“*emacs-source*”是存放 Emacs 源代码发行文件的那个目录。在《GNU Emacs Manual》（GNU Emacs 手册）里也能找到这份宣言。

自由编程联盟

如果你关注自由软件，就不会没有听说过自由编程联盟（League for Programming Freedom，简称 LPF）组织。这个联盟所关心的问题并不仅仅是自由软件，它针对的是对各种软件的可持续开发造成威胁的一切行为。自由编程联盟反对的是最近一个时期以来为软件申请专利和把命令语言版权化的不良趋势。正如这个联盟在它的一份时势分析报告里阐述的那样：它“与国会制定的法律制度（程序受版权保护）并不对立。我们的目标是扭转那些迎合特殊利益的法官们所造成的最新趋势，这些趋势通常极其明显地违背了国会的公众利益原则。”

LPF 认为设计和技术上而不是具体产品上的专利已经使软件的日常开发工作陷入了一种在法律上非常危险的境地，威胁到了程序员开发软件的自由。给技术方法（而非具体发明创造）颁发专利造成了极为严重和广泛的垄断。大家不妨设想一下，如果有人在 20 世纪 50 年代为链接列表（linked list，一种数据结构）申请了专利，那么今天的程序设计将如何发展。让软件开发人员去确定他们自己的设计和技术是否已经被申请专利不仅浪费时间和金钱，而且也极为困难。无论程序员多么聪明，

也不可能写出完全不依赖老技术的新软件，而它们当中的每一个都有可能已经被别人申请了专利。

事实上，软件专利和命令语言版权对一切软件开发工作都造成了威胁，但它们对自由软件的影响则可以用“致命”一词来形容。专利许可证中的条款与自由软件的传播方式是水火不相容的：比如说，如果软件自由传播、就不可能准确地统计出有多少副本正在流通，当然也就无法计算版税。你将无法把一个像 GNU Emacs 这样的程序转送给别人——除非你已经事先向这次交易所涉及的版权和专利拥有人全都做出了通报。更为严重的是，计算机上的公共服务程序有很多都是由自由软件的开发人员研制的；他们当中的很多人可能无法承受强加在他们正在开发的这类程序上的法律开支或软件许可税费。

LPF 在 1989 年 5 月和 1990 年 8 月曾经引起了社会的广泛关注。当时，为了抗议 Lotus Development Corporation（即中国读者比较熟悉的美国莲花软件公司，该公司的知名产品是 Lotus 1-2-3 电子表格程序）针对 Paperback Software, Inc. 公司就后者侵犯前者的操作界面版权而提起的法律诉讼，LPF 包围了 Lotus 软件公司在麻萨诸塞州剑桥市的总部。在这类被大众称之为“观感官司”的诉讼活动里，原告方坚持认为两种软件在用户操作界面方面的相似已经足以构成版权侵犯。Lotus 不仅打赢了这场官司，还引发了几场同样的诉讼，而且已经有几场官司又取得了相同的胜利。LPF 认为这种诉讼构成了垄断而且威胁到了软件工业的创造性——软件公司将被迫编写出一些与现有软件不相兼容的软件，而且将不愿意再对现有产品进行改进。

以下内容节选自 LPF 章程中阐明的奋斗目标：

1. 证明并引起公众的注意在各类计算机程序上存在的限制和垄断，而这种垄断阻止或限制了人们对特定种类的计算机程序的开发权。
2. 从公众利益出发，研究出必要的针对性办法和手段——包括教育、科研、公众集会、立法听证会以及干涉与公众利益问题有关的司法庭审（但不以法庭为针对目标）等，以有效地阻止、预防或限制这种垄断行为。
3. 支持并参加从事于和涉及上述活动的一切商务或其他活动，但这些活动必须是依麻萨诸塞州州法（General Law）第 180 章组建的公司能够依法实施的。

LPF 希望在财务方面能够得到大家的支持并邀请你成为它的会员。在我们写本书的时候，它的会员年费是：程序员、经理和教授每人 \$42 美元，学生每人 \$10.50 美元，

其他职业每人 \$21 美元。捐款和年费主要用于“补偿当事人、印制手持标语/示威牌/横幅等有可能劝说法庭、立法者和公众的东西”。LPF 每年还会向它的会员印发一些邮件和宣传品。它已经写出了两份时势分析报告，这些报告对有关事态的现状和应该采取的补救措施做了详细的论述。不过，这两份报告实在是太长了，我们没有办法把它们收录在这里；你可以通过向 *lpf@uunet.uu.net* 发电子邮件的办法来获得其 Texinfo 格式的文档。其他问题请致电 (617) 621-7084。

如果你愿意加入，请按下页表格中的要求提出申请；这份申请表是从该联盟的一份时势分析报告里摘抄下来的（译注 3）：

为了加入 LPF，请把一张支票和个人资料寄往：

League for Programming Freedom
1 Kendall Square #143
P. O. Box 9171
Cambridge, Massachusetts 02139

（如果在美国以外，请寄送一张在美国有分支机构的银行的美元支票，这可以节省我们的支票兑付费用。）

你的姓名：

你的邮寄地址：LPF 将按这个地址寄送邮件（每年几份）；请注明它是家庭地址还是工作地址。

你的工作单位和职务：

你的电话号码：请注明它是家庭电话还是工作电话。

你的电子邮件地址：便于我们联系你参加示威或写请愿信活动。（如果你不想我们联系你参加这类活动，请注明。但希望你能把电子邮件地址告诉我们，发电子邮件将节省我们的纸张和邮费。）

你是否有 LPF 可以用来加以宣传以增加公众印象的个人事项？比如说，如果你是或者曾经是一位大学教授、企业执行官，或者写出过有良好声誉的软件，请告诉我们。

你是否愿意为 LPF 的活动提供帮助？

译注 3：为方便中国读者，我们对这份申请表进行了翻译，但在格式上采取了符合中国人习惯的写法。它最终的解释权仍属于 League for Programming Freedom。

附录七

请支持自由软件基金会

即使有自愿者参加，软件开发和软件维护也是代价高昂的。我们必须告诉读者，而你们也应该知道：自由软件基金会并在真空里工作。他们的出版物一直在呼吁以下捐助：

- 计算机硬件
- 劳动（即你的时间）
- 金钱

我们无法预知FSF在某个特定的时期最需要哪些硬件或者劳动；如果你想贡献你的智慧或设备，请直接与他们联系。但作为劳动交换的一种抽象手段，FSF肯定需要支出金钱。如果你想做出捐助，请按下面的地址与他们联系：

Free Software Foundation, Inc.
59 Temple Place, Suite 330
Boston, MA 02111-1307
phone: (617) 542-5942
fax: (617) 542-2652
email: gnu@prep.ai.mit.edu
WWW: <http://twww1.vub.ac.be/studs/tw45639/fsf.htm>

附录八

Emacs 编辑

命令速查表

以下这些速查表都按用途进行了分类，表格的先后顺序与书中的讨论大致对应。但因为它们只是些速查表，所以不可能同时做到细致和全面；对一个像 Emacs 这样的规模宏大而又功能全面的文本编辑器来说，这不能不是一个遗憾。我们已经尽了最大的努力让这些速查表能够兼顾完备和简捷；但我们能够保证，即使我们犯了错误，也只错在“简捷”上而不是“完备”上。

表 H-1：对文件和编辑缓冲区进行操作（参见第一、四章）

键盘操作	命令名称	动作
emacs	(无)	在 shell 提示符处输入，启动 Emacs
C-x C-f <i>Files → Open File</i>	find-file	查找文件并打开它
C-x C-v	find-alternate-file	读入另外一个文件，替换掉用 “C-x C-f” 读入的文件
C-x i <i>Files → Insert File</i>	insert-file	把文件插入到光标的当前位置
C-x C-s <i>Files → Save Buffer</i>	save-buffer	保存文件
C-x C-w <i>Files → Save Buffer As</i>	write-file	把编辑缓冲区内容写入一个文件

表 H-1: 对文件和编辑缓冲区进行操作 (参见第一、四章) (续)

键盘操作	命令名称	动作
C-x C-c <i>Files → Exit Emacs</i>	save-buffers-kill-emacs	退出 Emacs
C-z	suspend-emacs	挂起 Emacs (用 exit 或 fg 命令唤醒)
C-x b	switch-to-buffer	移动到指定的编辑缓冲区
C-x C-b <i>Buffers → List All Buffers</i>	list-buffers	显示编辑缓冲区清单

表 H-2: 移动光标 (参见第二章)

键盘操作	命令名称	动作
C-f	forward-char	光标前移一个字符 (右)
C-b	backward-char	光标后移一个字符 (左)
C-p	previous-line	光标前移一行 (上)
C-n	next-line	光标后移一行 (下)
ESC f	forward-word	光标前移一个单词
ESC b	backward-word	光标后移一个单词
C-a	beginning-of-line	光标移到行首
C-e	end-of-line	光标移到行尾
C-v	scroll-up	屏幕上卷一屏
ESC v	scroll-down	屏幕下卷一屏
ESC <	beginning-of-buffer	光标前移到文件头
ESC >	end-of-buffer	光标后移到文件尾
C-l	recenter	重新绘制屏显画面, 当前行放在画面中心处

表 H-3: 删除文本、恢复文本、标记文本块 (参见第二章)

键盘操作	命令名称	动作
C-d	delete-char	删除光标位置上的字符
DEL	delete-backward-char	删除光标前面的字符
ESC d	kill-word	删除光标后面的单词
ESC DEL	backward-kill-word	删除光标前面的单词

表 H-3: 删除文本、恢复文本、标记文本块 (参见第二章) (续)

键盘操作	命令名称	动作
C-k	kill-line	从光标位置删除到行尾
C-w <i>Edit → Cut</i>	kill-region	删除文本块
ESC w 或 C-Insert <i>Edit → Copy</i>	kill-ring-save	复制文本块 (以便用 “C-y” 命令粘贴)
C-y 或 S-Insert <i>Edit → Paste Most Recent</i>	yank	恢复被删除的文本
ESC y	yank-pop	在用过 “C-y” 命令以后粘贴更早删除的文本
C-@ 或 C-SPACE	set-mark-command	标记文本块的开始(或结束)位置
C-x C-x	exchange-point-and-mark	互换光标和文本标记的位置

表 H-4: 命令的中止执行和编辑操作的撤销 (参见第二章)

键盘操作	命令名称	动作
C-g	keyboard-quit	放弃当前命令
C-x u	advertised-undo	撤销上一次编辑 (可以重复施用)
C-_ 或 C-/ <i>Edit → Undo</i>	undo	撤销上一次编辑
(无) <i>Files → Revert Buffer</i>	revert-buffer	把编辑缓冲区恢复到上次对文件进行存盘 (或者自动存盘) 时的状态

表 H-5: 交换文本位置和改变文本的大小写 (参见第二章)

键盘操作	命令名称	动作
C-t	transpose-chars	交换两个字符的位置
ESC t	transpose-words	交换两个单词的位置
C-x C-t	transpose-lines	交换两个文本行的位置
ESC c	capitalize-word	把单词的首字母改为大写
ESC u	upcase-word	把单词的字母全部改为大写
ESC l	downcase-word	把单词的字母全部改为小写

表 H-6: 查找与替换 (参见第三章)

键盘操作	命令名称	动作
C-s	isearch-forward	向文件尾方向开始递增查找操作
C-r	isearch-backward	向文件头方向开始递增查找操作
RETURN	(无)	退出一次成功的查找操作
C-g	keyboard-quit	取消递增查找操作 (可能需要连接两次这个组合键)
DEL	(无)	删除查找字符串中不正确的字符
C-s RETURN <i>Search → Search</i>	(无)	向文件尾方向开始非递增查找操作
C-r RETURN <i>Search → Search Backwards</i>	(无)	向文件头方向开始非递增查找操作
ESC % <i>Search → Query-Replace</i>	query-replace	进入查询 - 替换

表 H-7: 查询 - 替换操作中使用的命令 (参见第三章)

键盘操作	动作
SPACE 或 y	替换并前进到下一个位置
DEL 或 n	不替换: 前进到下一个位置
.	在当前位置做替换后退出操作
,	替换并暂停 (按空格键或“y”继续)
!	对其余全部进行替换, 不再要求询问
^	返回前一次进行了替换的位置
RETURN	退出查询 - 替换操作

表 H-8: 拼写和简写词汇 (参见第三章)

键盘操作	命令名称	动作
ESC \$ <i>Edit → Spell → Check Word</i>	ispell-word	检查光标位置上的单词或者光标后面的单词
(无) <i>Edit → Spell → Check Region</i>	ispell-region	检查文本块里的单词

表 H-8：拼写和简写词汇（参见第三章）（续）

键盘操作	命令名称	动作
(无)	spell-word	检查光标位置上的单词或者光标后面的单词
(无)	spell-buffer	检查当前编辑缓冲区的拼写
(无)	abbrev-mode	进入（或退出）单词简写模式
C-x a - 或 C-x a i g	inverse-add-global-abbrev	输入全局性简写词之后，输入其短语定义
C-x a l l	inverse-add-local-abbrev	输入局部性简写词之后，输入其短语定义
(无)	write-abbrev-file	保存简写词汇文件
(无)	list-abbrevs	查看简写词汇表

表 H-9：窗口（参见第四章）

键盘操作	命令名称	动作
C-x 2 <i>Files → Split Window</i>	split-window-vertically	把当前窗口分割为上、下排列的两个窗口
C-x o	other-window	移动到其他窗口；如果有多个窗口，按顺时针方向移动到下一窗口
C-x 0	delete-window	删除当前窗口
C-x 1 <i>Files → One Window</i>	delete-other-windows	删除所有窗口，只保留当前窗口
C-x ^	enlarge-window	加高当前窗口
(无)	shrink-window	压低当前窗口
ESC C-v	scroll-other-window	对其他窗口做卷屏操作
C-x 4 f	find-file-other-window	在其他窗口里查找并打开一个文件

表 H-10: 窗格 (参见第四章)

键盘操作	命令名称	动作
C-x 5 o <i>Files → Make New Frame</i>	other-frame	移动到其他窗格
C-x 5 2 <i>Files → Make New Frame</i>	make-frame	创建一个新窗格
C-x 5 0 <i>Files → Delete Frame</i>	delete-frame	删除当前窗格
C-x 5 f	find-file-other-frame	在一个新窗格里查找文件
C-x 5 b	switch-to-buffer-other-frame	创建新窗格并显示另一个编辑缓冲区

表 H-11: 书签 (参见第四章)

键盘操作	命令名称	动作
C-x r m <i>Search → Bookmarks → Set bookmark</i>	bookmark-set	在当前光标位置处设置一个书签
C-x r b <i>Search → Bookmarks → Jump to bookmark</i>	bookmark-jump	跳转到书签指示的位置

表 H-12: 发送邮件 (参见第六章)

键盘操作	命令名称	动作
C-x m	mail	打开 “*mail*” 编辑缓冲区
C-x 4 m	mail-other-window	在一个新窗口里打开 “*mail*” 编辑缓冲区
C-x 5 m	mail-other-frame	在一个新窗格里打开 “*mail*” 编辑缓冲区
C-c C-f C-t <i>Headers → To</i>	mail-to	移动到 “To:” 栏
C-c C-f C-c <i>Headers → Cc</i>	mail-cc	移动到 “Cc:” 栏 (如果没有就创建之)
C-c C-f C-s <i>Headers → Subject</i>	mail-subject	移动到 “Subject:” 栏

表 H-12：发送邮件（参见第六章）（续）

键盘操作	命令名称	动作
C-c C-w <i>Mail → Insert Signature</i>	mail-signature	插入 “.signature” 文件的内容
C-c C-c <i>Mail → Send Message</i>	mail-send-and-exit	发送邮件并退出 “*mail*” 编辑缓冲区
(无)	define-mail-alias	为某个名字或邮件表定义一个缩写形式

表 H-13：读取邮件（参见第六章）

键盘操作	命令名称	动作
(无) <i>Tools → Read Mail</i>	rmail	启动 RMAIL 读取邮件
SPACE	scroll-up	卷屏，查看此消息的下一个画面
DEL	scroll-down	卷屏，查看此消息的上一个画面
.	rmail-beginning-of-message	移动到此消息的开头
n <i>Move → Next</i>	rmail-next-undeleted-message	移动到下一条消息
p <i>Move → Previous</i>	rmail-previous-undeleted-message	移动到上一条消息
< <i>Move → First</i>	rmail-first-message	移动到第一条消息
> <i>Move → Last</i>	rmail-last-message	移动到最后一条消息
j	rmail-show-message	如果这个命令的前面有一个数字 “n”，跳到第 n 条消息
d <i>Delete → Delete</i>	rmail-delete-forward	给邮件加上待删除标记，然后移动到下一个
C-d	rmail-delete-backward	给邮件加上待删除标记，然后移动到上一个

表 H-13: 读取邮件 (参见第六章) (续)

键盘操作	命令名称	动作
u <i>Delete → Undelete</i>	rmail-undelete-previous-message	去掉邮件消息上的待删除标记
x <i>Delete → Expunge</i>	rmail-expunge	删除已经加有待删除标记的全部消息
o filename RETURN <i>Classify → Output (Rmail)</i>	rmail-output-to-rmail-file	把邮件消息保存为 RMAIL 文件格式
C-o filename RETURN <i>Classify → Output (inbox)</i>	rmail-output	把邮件消息保存为 UNIX 邮件文件格式 (一个标准的 ASCII 文本文件)

表 H-14: RMAIL 邮件清单 (参见第六章)

键盘操作	命令名称	动作
h	rmail-summary	显示邮件清单
d	rmail-summary-delete-forward	给消息加上待删除标记 (在消息序号前出现字母 “D” 标记)
u	rmail-summary-undelete	去掉当前消息上的待删除标记
n	rmail-summary-next-msg	移动到下一条消息并把它显示在 RMAIL 窗口里
p	rmail-summary-previous-msg	移动到上一条消息并把它显示在 RMAIL 窗口里
x	rmail-summary-expunge	删除所有加有待删除标记的消息
q	rmail-summary-quit	退出 RMAIL

表 H-15: shell 模式命令 (参见第五章)

键盘操作	命令名称	动作
(无)	shell	进入 shell 模式
C-c C-c <i>Signals → Break</i>	comint-interrupt-subjob	中断当前作业; 相当于 UNIX 的 shell 中的 “C-c” 组合键

表 H-15: shell 模式命令 (参见第五章) (续)

键盘操作	命令名称	动作
C-d C-c C-d <i>Signals → EOF</i>	comint-delchar-or-maybe-eof comint-send-eof	如果是在编辑缓冲区的末尾, 送出 EOF 字符; 如果是在其他位置, 删除一个字符 送出 EOF 字符
C-c C-u	comint-kill-input	删除当前行; 相当于 UNIX 的 shell 中的 “C-u” 组合键
C-c C-z <i>Signals → STOP</i>	comint-stop-subjob	对非 X 用户, 挂起或者停止一个作业; 相当于 UNIX 的 shell 中的 “C-z” 组合键
ESC p <i>In/Out → Previous Input</i>	comint-previous-input	检索上一个命令 (可以重复执行以找回更早的命令)
ESC n <i>In/Out → Next Input</i>	comint-next-input	检索下一个命令 (可以重复执行以找回更近的命令)
RETURN	comint-send-input	送出当前行上的输入
TAB <i>Complete → Complete Before Point</i>	comint-dynamic-complete	自动补足当前命令、文件名或者变量名
C-c C-o <i>In/Out → Kill Current Output Group</i>	comint-kill-output	删除最后一条命令的输出
C-c C-e <i>In/Out → Show Maximum Output</i>	comint-show-maximum-output	把输出内容的最后一行移到窗口的底部

表 H-16: 用来创建正则表达式的字符 (参见第三章)

字符	匹配情况
^	匹配行首
\$	匹配行尾
.	匹配任意单个字符 (类似于文件名中的问号?)

表 H-16: 用来创建正则表达式的字符 (参见第三章) (续)

字符	匹配情况
<code>.</code>	匹配任意 (零个以上) 个字符
<code>\<</code>	匹配单词的开头
<code>\></code>	匹配单词的结尾
<code>[]</code>	匹配方括号中的任何一个字符; 比如 “[a-z]” 表示匹配任意一个字母表字符

表 H-17: Dired 命令 (参见第五章)

键盘操作	命令名称	动作
C-x d <i>Files → Open Directory</i>	dired	启动 Dired
C <i>Operate → Copy to</i>	dired-do-copy	复制文件
d <i>Mark → Flag</i>	dired-flag-file-deletion	给文件加上待删除标记
f <i>Immediate → Find This File</i>	dired-find-file	编辑文件
g <i>Files → Revert Buffer</i>	revert-buffer	从磁盘上重新读入目录
n <i>Subdir → Next Dirline</i>	dired-next-line	移动到下一行
q	dired-quit	退出 Dired
R <i>Operate → Rename to</i>	dired-do-rename	重新命名文件
u <i>Mark → Unmark</i>	dired-unmark	去掉待操作标记
x	dired-do-flagged-delete	删除加有待删除标记 “D”的文件
Z <i>Operate → Compress</i>	dired-do-compress	对文件进行压缩或解压缩操作

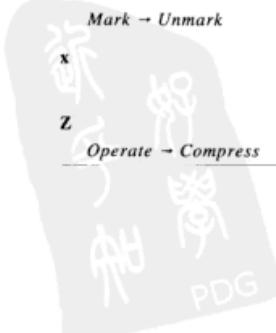


表 H-17: **Dired** 命令 (参见第五章) (续)

键盘操作	命令名称	动作
ESC DEL	dired-unmark-all-files	把所有文件上的各种待操作标记 (不管它是什么) 都去掉
>	dired-next-dirline	移动到下一个目录
<	dired-prev-dirline	移动到上一个目录
s	dired-sort-toggle-or-edit	对 Dired 画面按日期或按文件名进行排序 (在两者之间切换)

表 H-18: **nroff** 模式 (参见第九章)

键盘操作	命令名称	功能
(无)	nroff-mode	进入 nroff 模式
ESC n	forward-text-line	把光标移动到下一个文本行
ESC p	backward-text-line	把光标移动到上一个文本行
ESC ?	count-text-lines	统计文本块中的文本行数
(无)	electric-nroff-mode	进入一个副编辑模式。在这个模式里, 若输入必须配对出现的 nroff 命令组中的第一个, 按“ C-J ”组合键, Emacs 就会自动插入该配对命令组中的第二个命令
C-j	electric-nroff-newline	只能用在 nroff 配对模式里, 自动插入一组必须配对出现的 troff 宏定义标记中的第二个

表 H-19: **TeX** 模式 (参见第九章)

键盘操作	命令名称	功能
(无)	tex-mode	根据文件内容进入 TeX 模式或 LATEX 模式
(无)	plain-tex-mode	进入 TeX 模式
(无)	latex-mode	进入 LATEX 模式

表 H-19: TeX 模式 (参见第九章) (续)

键盘操作	命令名称	功能
C-j	tex-terminate-paragraph	插入两个硬回车 (标准的段落结束标志) 并检查段落的语法
C-c {	tex-insert-braces	插入两个花括号, 然后把光标放在它们中间
C-c }	up-list	如果光标在两个花括号中间, 就把光标放到紧跟在右括号后面的位置上
(无) <i>TeX → Validate Buffer</i>	validate-tex-buffer	检查编辑缓冲区有无语法错误
C-c C-b <i>TeX → TeX Buffer</i>	tex-buffer	对编辑缓冲区进行 TeX 或 LATEX 排版处理
C-c C-l <i>TeX → TeX Recenter</i>	tex-recenter-output-buffer	把排版信息画面显示在屏幕上, 显示 (至少) 最近一条出错信息
C-c C-r <i>TeX → TeX Region</i>	tex-region	对文本块进行 TeX 或 LATEX 排版处理
C-c C-k <i>TeX → TeX Kill</i>	tex-kill-job	中断 TeX 或 LATEX 的排版处理工作
C-c C-p <i>TeX → TeX Print</i>	tex-print	打印 TeX 或 LATEX 的排版输出
C-c C-q <i>TeX → Show Print Queue</i>	tex-show-print-queue	查看打印队列
C-c C-e	tex-close-latex-block	只能用在 LATEX 模式里, 自动插入某个命令组中的第二个命令

表 H-20: 宏 (参见第十章)

键盘操作	命令名称	动作
C-x (start-kbd-macro	开始录制一个宏
C-x)	end-kbd-macro	结束宏的录制工作
C-x e	call-last-kbd-macro	执行最近一次录制的宏
C-u C-x (universal-argument; start-kbd-macro	执行最近一次录制的宏, 然后允许再增加新按键组合

表 H-20: 宏 (参见第十章) (续)

键盘操作	命令名称	动作
C-x q	kbd-macro-query	在宏定义里插入一个查询
C-u C-x q	(无)	在宏定义里插入一个递归编辑
ESC C-e	exit-recursive-edit	退出递归编辑

表 H-21: Gnus 新闻阅读器 “Group” 编辑缓冲区操作命令 (参见第六章)

键盘操作	命令名称	动作
(无)	gnus	启动 Gnus
SPACE	gnus-group-read-group	阅读光标位置处的新闻组里的文章
u	gnus-group-unsubscribe-current-group	订阅或者撤销订阅这个新闻组
j	gnus-group-jump-to-group	提示输入一个新闻组名称以转入其中 (可以转到未曾订阅的新闻组)
l	gnus-group-list-groups	列出已订阅并且有新闻可读的新闻组
A k	gnus-group-list-killed	列出那些被 “.newsr.eld” 文件里的语句所排除掉的新闻组
L	gnus-group-list-all-groups	列出这个服务器上全部可用的新闻组
g	gnus-group-get-new-news	取回启动 Gnus 以来新收到的新闻
a	gnus-group-post-news	为这个新闻组写一篇新文章
C-x C-t	gnus-group transpose-groups	交换当前行和上一行的位置
q	gnus-group-exit	退出新闻功能并刷新 “.newsr” 文件

表 H-22: Gnus 新闻阅读器 “Summary” 编辑缓冲区操作命令 (参见第六章)

键盘操作	命令名称	动作
SPACE	gnus-summary-next-page	文章前卷
DEL	gnus-summary-prev-page	文章后卷
l	gnus-summary-goto-last-article	移动到刚读过的最后一篇文章 (如果移动得太快, 可以用这条命令返回)

表 H-22: Gnus 新闻阅读器 “Summary” 编辑缓冲区操作命令 (参见第六章) (续)

键盘操作	命令名称	动作
H f	gnus-summary-fetch-faq	取回这个新闻组的常见问题答疑文件
n	gnus-summary-next-unread-article	移动到下一篇新闻
p	gnus-summary-prev-unread-article	移动到上一篇新闻
u	gnus-summary-tick-article-forward	给当前文章加上未阅读标记，在它旁边放上一个惊叹号以保留它供今后阅读。如果重复输入此命令，则保存下一篇文章
C-o	gnus-summary-save-article-mail	以 UNIX 格式保存当前文章
o	gnus-summary-save-article	以 RMAIL 格式保存当前文章
q	gnus-summary-exit	返回 “Group” 编辑缓冲区
d	gnus-summary-mark-as-read-forward	给当前文章加上已阅读标记，从当前行开始向下移动
C-c C-f	gnus-summary-mail-forward	把这篇文章的副本发送给某人
g	gnus-summary-show-article	显示当前文章（特别适用于扩展了“Summary”窗口或者需要在这个窗口里移动的情况）
x	gnus-summary-remove-lines-marked-as-read	把带有已阅读标记的文章全部删掉
=	gnus-summary-expand-window	扩展 “Summary” 窗口，让它充满整个屏幕

表 H-23: Telnet (参见第七章)

键盘操作	命令名称	动作
(无)	telnet	进入 Telnet 模式
C-d	comint-delchar-or-maybe-eof	根据上下文，发送 EOF 字符或删除光标位置下的字符
RETURN	telnet-send-input	送出 Telnet 输入
C-c C-c	telnet-interrupt-subjob	中断当前作业：相当于 shell 中的 “C-c” 组合键

表 H-23: Telnet (参见第七章) (续)

键盘操作	命令名称	动作
C-c C-q	send-process-next-char	发送紧随其后的控制字符; 相当于 shell 中的“C-q”组合键
C-c C-d	comint-send-eof	发送 EOF (文件尾) 字符
C-c C-r	comint-show-output	让输出内容的第一行显示在窗口的顶部
C-c C-e	comint-show-maximum-output	让输出内容的最后一行显示在窗口的底部
C-c C-o	comint-kill-output	删除上一个命令的输出内容
C-c C-z	telnet-c-z	挂起或暂停这个作业; 相当于 shell 中的“C-z”组合键
C-c C-u	comint-kill-input	删除当前行; 相当于 shell 中的“C-u”组合键
ESC n	comint-next-input	查看此后输入的命令 (重复此操作可查看到更靠后的命令)
ESC p	comint-previous-input	查看此前输入的命令 (重复此操作可查看到更靠前的命令)

表 H-24: C 模式 (参见第十二章)

键盘操作	命令名称	动作
ESC C-a	beginning-of-defun	移动到当前函数的开头
ESC C-e	end-of-defun	移动到当前函数的结尾
ESC C-h	mark-c-function	把光标放到函数的开头, 把文本块标记放到函数的结尾

表 H-25: LISP 模式 (参见第十二章)

键盘操作	命令名称	动作
ESC C-b	backward-sexp	移动到上一个 S- 表达式
ESC C-f	forward-sexp	移动到下一个 S- 表达式
ESC C-t	transpose-sexps	交换光标前后的两个 S- 表达式的位置
ESC C-@	mark-sexp	把文本块标记设置在当前 S- 表达式的末尾, 把光标设置在当前 S- 表达式的开头

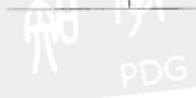


表 H-25: LISP 模式 (参见第十二章) (续)

键盘操作	命令名称	动作
ESC C-k	kill-sexp	删除光标后面的那个 S- 表达式
ESC C-DEL	backward-kill-sexp	删除光标前面的那个 S- 表达式
ESC C-n	forward-list	移动到上一个列表
ESC C-p	backward-list	移动到下一个列表
ESC C-d	down-list	向前移动、进入下一级括号层次
(无)	up-list	向前移动、退出当前的括号层次
ESC C-u	backward-up-list	向后移动、退出当前的括号层次
ESC C-a	beginning-of-defun	移动到当前函数的开头
ESC C-e	end-of-defun	移动到当前函数的结尾
ESC C-h	mark-defun	把光标放到函数的开头，把文本块标记放到函数的结尾

表 H-26: FORTRAN 模式 (参见第十二章)

键盘操作	命令名称	动作
C-c C-n	fortran-next-statement	向前移动一个语句
C-c C-p	fortran-previous-statement	向后移动一个语句
ESC C-a	beginning-of-fortran-subprogram	移动到当前子程序的开头
ESC C-e	end-of-fortran-subprogram	移动到当前子程序的结尾
ESC C-h	mark-fortran-subprogram	把光标放到子程序的开头，把文本块标记放到子程序的结尾

表 H-27: Emacs 中的鼠标操作 (参见第十四章)

鼠标动作	命令名称	对应的 Emacs 标准命令 (和组合键)
左键	x-mouse-set-point	(把 Emacs 光标移动到鼠标光标位置)
中键	x-paste-text	yank ("C-y")
右键	x-cut-text	copy-region-as-kill ("ESC w")
C- 中键	x-cut-and-wipe-text	kill-region ("C-w")

表 H-27: Emacs 中的鼠标操作（参见第十四章）(续)

鼠标动作	命令名称	对应的 Emacs 标准命令 (和组合键)
C- 右键	x-mouse-select-and-split	split-window-vertically ("C-x 2")
S- 中键	x-cut-text	copy-region-as-kill ("ESC w")
S- 右键	x-paste-text	yank ("C-y")
C-S- 右键	x-mouse-keep-one-window	delete-other-windows ("C-x 1")

表 H-28: 在线帮助系统 (参见第十六章)

键盘操作	命令名称	回答的问题
C-h c	describe-key-briefly	这个按键组合将运行哪个命令
C-h k	describe-key	这个按键组合将运行哪个命令？这个命令的作用是什么
C-h l	view-lossage	最近输入的 100 个字符是什么
C-h w	where-is	这个命令的按键绑定是什么
C-h f	describe-function	这个函数的作用是什么
C-h v	describe-variable	这个变量的含义是什么？它有哪些可取值
C-h m	describe-mode	查看当前编辑缓冲区所在编辑模式的有关资料
C-h b	describe-bindings	这个编辑缓冲区都有哪些按键绑定
C-h a	command-apropos	这个概念都涉及到哪些命令
C-h t	help-with-tutorial	运行 Emacs 教程
C-h i	info	启动文档阅读器 Info 程序

表 H-29: Emacs 重要的编辑模式

模式	功能
基本模式 (fundamental-mode)	默认模式，无特殊行为
文本模式 (text-mode)	主编辑模式：书写文字材料 (参见第二章)
缩进文本模式 (indented-text-mode)	主编辑模式：自动缩进文本 (参见第八章)
C 模式 (c-mode)	主编辑模式：书写 C 语言程序 (参见第十二章)



表 H-29: Emacs 重要的编辑模式 (续)

模式	功能
FORTRAN 模式 (fortran-mode)	主编辑模式: 书写 FORTRAN 程序 (参见第十二章)
LISP 模式 (lisp-mode)	主编辑模式: 书写 LISP 程序 (参见第十二章)
RMAIL 模式 (rmail-mode)	主编辑模式: 阅读和组织电子邮件 (参见第六章)
nroff 模式 (nroff-mode)	主编辑模式: 按 nroff 的要求对文件进行排版 (参见第九章)
TEX 模式 (tex-mode)	主编辑模式: 按 TeX 的要求对文件进行排版 (参见第九章)
LATEX 模式 (latex-mode)	主编辑模式: 按 LATEX 的要求对文件进行排版 (参见第九章)
大纲模式 (outline-mode)	主编辑模式: 书写大纲 (参见第八章)
shell 模式 (shell-mode)	主编辑模式: 在 Emacs 里运行一个 UNIX shell (参见第五章)
缩写模式 (abbrev-mode)	副编辑模式: 缩写单词 (参见第三章)
自动换行模式 (auto-fill-mode)	副编辑模式: 激活字换行 (word wrap) 功能 (参见第二章)
改写模式 (overwrite-mode)	副编辑模式: 打字时对字符进行改写而不是插入 (参见第二章)
自动保存模式 (auto-save-mode)	副编辑模式: 按一定周期自动把文件保存到特殊的自动保存文件里 (参见第二章)
行号模式 (line-number-mode)	副编辑模式: 在状态行上显示当前的行编号 (参见第二章)
临时标记模式 (transient-mark-mode)	副编辑模式: 反显被选取区域 (仅限 X 界面) (参见第二章)



词汇表

abbrev mode (简写词汇模式)

用来定义简写词汇的编辑模式，由用户定义的简写词汇在输入的时候被自动替换。可以为短语、长单词或容易拼写错误的单词定义简写词汇。Emacs的简写词汇功能（vi 编辑器里也有类似功能）类似于 Microsoft Word 中的自动更正功能，但比后者出现得要早得多。关于简写词汇模式的详细讨论请参考第三章。

ange-ftp mode (ange-ftp 模式)

一个很容易使用的FTP (File Transfer Protocol, 文件传输协议) 接口，作者是Andy Norman。用户可以用 **find-file** 命令打开被保存在因特网或者其他网络上的文件或目录，就像它们是在本地系统上一样。要想指定远程文件，请输入 “`/user@systemname:/ pathToFile/filename`”。最开始的斜线字符和系统名与路径名之间的冒号很容易忘记；但缺少了它们，ange-ftp 模式将不会工作。如果省略了路径名和文件名，Emacs 将使用 Dired 来显示远程系统上的顶级目录。无须使用 FTP 命令来检索文件，它们完全能够用 Dired 命令来显示和复制。因此，ange-ftp 模式很适合用来查看和下载远程文件。从 Emacs 19.29 开始，ange-ftp 模式已经包括在 Emacs 的发行版本里了。

auto-fill mode (自动换行模式)

Emacs 实现自动换行功能的一个副编辑模式。在自动换行模式下，当到达文本行尾的时候可以继续进行输入，Emacs 会把换行符自动插入到适当位置。自动换行模式在缺省状态是关闭的。

auto-save file (自动保存文件, #file#)

Emacs 会定期把用户的编辑缓冲区保存到一种叫做“自动保存 (auto-save)”文件的临时文件里。在 Emacs 会话被非正常终止的时候，Emacs 也会把文件保存到自动

保存文件里。比如，如果系统在用户正编辑着一个名为 *budget* 的文件时死机了，那么用户可以在系统恢复正常后找到一个名为 #*budget*# 的自动保存文件。关于自动保存文件的详细讨论请参考第二章。

backup file (备份文件, *file*)

在 Emacs 保存文件的时候，它会先把当前版本复制到一个名字相同但后面多出一个波浪字符 (~) 的文件里。比如说，如果保存的文件是 *budget*，Emacs 会把前一个版本转移到文件 *budget*~ 里。此后，如果用户决定不想保留自己所做的修改，就可以使用备份文件。关于备份文件的详细讨论请参考第二章。

body (消息体/段落体)

1. 邮件消息分为消息头 (header，里面的信息用来指明谁是收信人和谁是发信人) 和消息的内容两部分，后者就是消息体。2. 在大纲模式里，编辑缓冲区里的内容被分为标题 (大纲的框架) 和文本段落，后者构成了段落体。

bookmark (书签)

文件中一个有名字的位置。打开文件操作永远会把用户带到第一行；用书签打开文件，会把用户带到指定的位置。书签特别适合用在大纲文档等想在下次打开时直接回到上次离开位置处的文件里。关于书签的详细讨论请参考第四章。

browser (浏览器)

1. 一个允许用户访问 WWW 网的程序。它又分为图形化浏览器，比如 Mosaic 和 Netscape，和文本浏览器，比如 Lynx 和 W3 模式。2. 查看某种格式的信息所使用的阅读器程序。Info 模式就是一种用来阅读 Info 格式文档的浏览器。

buffer (编辑缓冲区)

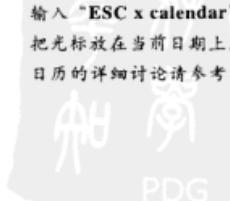
屏幕上 Emacs 显示文件的一个副本或者空白区域以供用户进行输入的工作区。当用户编辑文件的时候，Emacs 会把文件复制到一个同名的编辑缓冲区里；而磁盘上的文件则不会被修改。当保存编辑缓冲区的时候，Emacs 会把用户做的修改复制到磁盘上文件里。除对应于文件的编辑缓冲区外，Emacs 本身还会在阅读和书写电子邮件、配合 UNIX shell 工作以及使用日历等工作中创建专用的编辑缓冲区。

byte-compile (字节编译)

编译 Emacs LISP 文件（以 “.el” 为后缀名）的一种方法。对文件进行字节编译之后，它的后缀名字将变成 “.elc”。经过字节编译的 Emacs LISP 函数加载和运行得更快。

calendar (日历)

输入 “**ESC x calendar**”，Emacs 就将在屏幕底部显示一个三个月的日历，并且会把光标放在当前日期上。在日历里，可以用日记条目标记日期和查看节假日。关于日历的详细讨论请参考第五章。



clipboard (剪贴板)

参见 kill ring (删除环)。

comment (注释)

程序设计语言或文本排版程序不进行处理的文本。Emacs会在用户按下“**ESC**”时，根据用户所在的编辑模式自动插入注释语法。文本模式和基本模式等普通的编辑模式里没有定义注释语法。

completion (自动补足功能)

一种快捷输入方式：用户输入命令、变量或文件名的前几个字符再按下**TAB**键，Emacs就会自动补足它(如果它是唯一的)或者在一个“*Help*”窗口里列出候选。

copyleft

一种软件必须自由共享的共识，使接受方也能够按此方式共享这个软件。**copyright**(版权)限制了信息的使用，而根据附录四中自由软件基金会(Free Software Foundation, FSF)的条款，copyleft将保证软件能够继续被任何人自由共享。术语“copyleft”还被用来引述《GNU通用公共许可证》。

cursor (光标)

光标指示了在编辑缓冲区里的特定位置。有时候，Emacs把光标也称为插入点。从技术上讲，插入点位于光标和前一个字符的中间。

customization (定制)

在缺省的情况下，Emacs有一套固定的行为方式。通过定制，可以把这种固定的行为方式改变为自己需要和喜欢的形式。对Emacs的定制是通过“*.emacs*”文件进行的。关于定制的详细讨论请参考第十一章。

cut (剪切操作)

删除文本，以便用于粘贴。

default (缺省值)

变量或选项的默认取值。比如说，在缺省的情况下，自动换行模式是关闭的。

default direction (缺省绘制方向)

在输入文本的时候，文本通常是从左向右出现的。在图形模式里，可移动的方向有8个。刚进入图形模式时，缺省绘制方向是右。如果指定了一个不同的方向——比如上、下、左上、右下等等，它就将成为缺省绘制方向。“**C-e C-f**”组合键的向前移动和“**C-c C-b**”组合键的向后移动都是相对缺省绘制方向而言的。

delete (删除操作)

删除文本且不把它保存在删除环里以供今后粘贴等操作使用的操作。

diary (日记)

Emacs允许为特定日期加注留言的功能。此后，Emacs将在特定日期到来时显示一

个提醒。日记条目可以设置在给定的日期、日期块或循环周期（比如每两星期一循环）上。关于日记的详细讨论请参考第五章。

digest (邮件文摘)

为求方便而单独做为一条消息发送的一组邮件消息。来自邮件表的消息经常是以邮件文摘的形式发送的。如果使用 RMAIL 阅读邮件，则命令 “**ESC x undigestify-rmail-message**” 将把邮件文摘分割为各个消息。在 Gnus 里，可以用 “**C-d**” 组合键把做为一个新闻组稿件的邮件文摘扩展开来。

Dired

Emacs 的目录编辑器。可以通过 Dired 完成文件和目录上的各种操作，比如移动、压缩、删除、复制和字节编译等。关于 Dired 的详细讨论请参考第五章。

dot (插入点)

光标位置或插入点的同义词。在 Emacs 的在线帮助功能里会见到 “dot” 这个术语。

.emacs file (.emacs文件)

用来改变 Emacs 缺省行为的初始化文件。这个文件里的命令将在启动 Emacs 的时候运行。但 Emacs 没有这个文件也完全能够运行。我们在这本书里介绍了许多可以添加到 “.emacs” 文件里以改变 Emacs 某些行为的语句。

email (电子邮件)

电子化的邮件。Emacs 自己带有一个名为 RMAIL 的邮件阅读程序和一个名为 sendmail（不要把它与 UNIX 的 sendmail 命令混为一谈）的邮件发送程序。我们在第六章里对它们进行了讨论。Emacs 还有面向邮件发送程序 MH 和 vm 的接口。对 mh 接口感兴趣的读者可以读一读 Jerry Peek 写的《MH & xmh: Email for Users and Programmers》。

file (文件)

磁盘上的一个存储区域。当用 Emacs 打开一个文件的时候，它会把文件复制到一个编辑缓冲区，即屏幕上的一个工作区域里。当保存修改的时候，Emacs 又会把当前编辑缓冲区里的修改以改写方式复制回磁盘上的文件。

fill prefix (缩进前缀)

在文本模式和自动换行模式里，缩进前缀是定义并让 Emacs 插入到段落中的每一行开头去的一串字符。缩进前缀可以是一串空格（便于对文本进行缩进）或者是一个简单的大于号 (>)；后者通常用在电子邮件里区分引用性文字和上下段落。

flow control (流控制)

终端与主机进行通信的一种方式。有些计算机把 “C-s” 解释为停止接受输入，把 “C-q” 解释为重新开始接受输入。但 “C-s” 和 “C-q” 在 Emacs 里有其他的用途，如果将 “C-s” 用做流控制字符，终端就会在用户输入带 “C-s”的 Emacs 命令的时

候被挂起来。可以通过“**ESC x enable-flow-control**”命令用“C-V”和“C-^”分别来代替 Emacs 命令中的“C-s”和“C-q”。关于流控制的详细讨论请参考第十一章。

formfeed (换页符)

一个控制字符，通常显示为“^L”，它的作用是让打印机从新的一页开始打印。如果想在 Emacs 文件里插入一个 formfeed 换页符，请按下“**C-q C-l**”组合键。

frame (窗格)

显示一个 Emacs 编辑缓冲区的 X 窗口。在缺省的情况下，X 窗口系统的用户只有一个窗格，但他们能够在一次 Emacs 会话里创建多个 X 窗口。在只有一个窗格的时候，系统名将显示为窗口的标题；在有多个窗格的时候，窗格的名字将是编辑缓冲区的名字。关于窗口和窗格的详细讨论请参考第四章。（字符终端上也能使用窗格，但它们在窗口化的操作界面里更有用）。

global abbreviations (全局简写词汇)

由用户定义的能够用在各种主编辑模式下的简写词汇。与此形成对比的是局部简写词汇，它们只能用在对它们做出定义的编辑模式里。因此，全局简写词汇适合用在想让 Emacs 进行自动替换的时候，而局部简写词汇适合用在只想在文本模式而不想用在 C 模式里的替换。关于简写词汇的详细讨论请参考第四章。

GNU

“GNU's Not UNIX (GNU 不是 UNIX)” 的字头缩写。

Gnus

GNU Emacs 19 自带的一个新闻阅读器。Gnus 使 Emacs 用户能够在不退出 Emacs 的情况下阅读和投稿 Usenet 新闻。

header (消息头/控制头)

1. 在 L^AT_EX 等文件的开头对文档格式某些特征做出定义的部分。2. 在电子邮件里，在邮件消息的开头给出发信人、收信人、发信日期以及其他信息的标题部分。3. 嵌在文件里的版本控制信息，它们会在签出、签入和复原操作中自动刷新。

home directory (主目录)

个人目录。在一台 UNIX 系统上，它往往就是子目录 /home /username。

home page (主页)

1. 启动 WWW 浏览器时看到的缺省页面。2. 在某给定 WWW 站点上看到的第一个页面。

hotlink (热链)

超链接的另一个术语。

hotlist (收藏夹)

WWW 上用户最喜欢访问的站点的超链表。

hyperlink (超链接)

文本或图形中指向某个本地或远程资源的链接。选择一个超链接将会进入指定的资源。

Hypertext Markup Language (HTML, 超文本标记语言)

超文本标记语言，一种对文档按 WWW 显示要求进行排版的 ASCII 标记语言。

HTML 文件也叫做 Web 页面，可以用一种人们称之为浏览器的查看器进行浏览。

incremental search (递增查找)

Emacs 中的一种查找方式，输入第一个字符时，查找工作就开始了。

initialization file (初始化文件)

参见 “*.emacs*” 文件。

kill (删除操作)

删除并保存到删除环。

kill file (过滤文件)

一个由新闻阅读器（比如 Gnus）读取的文件，它将根据用户设置的规则排除用户不想看到的消息。比如说，过滤文件可以把某特定用户投稿的消息或者关于某特定主题的消息过滤掉。

kill ring (删除环)

Emacs 存放被删除文本的地方。在缺省的情况下，删除环可以容纳最近 30 次删除或剪切下来的东西。但有些删除操作（比如字符删除命令 “C-d”）不会把被删除的文本保存到删除环里。关于删除环的详细讨论请参考第二章中的讨论。

local abbreviations (局部简写词汇)

用户定义的仅能用在某个指定编辑模式里（即仅供编辑模式局部使用）的简写词汇。比如说，用户可能想让文本模式和 C 模式分别使用不同的简写词汇。关于局部简写词汇的详细讨论请参考第三章。

macros (宏)

一组被录制下来的按键序列或鼠标动作，目的是减少重复性的编辑工作。关于宏的详细讨论请参考第十章。

major mode (主编辑模式)

Emacs 用来使其行为适应手中主要任务的方式。文本模式是主要用于书写文字材料的主编辑模式；C 模式是主要用于书写 C 语言代码的主编辑模式。不同的主编辑模式有不同的编辑命令，这些命令只对该主编辑模式有意义；那些能用在一切编辑模式里的命令叫做全局编辑命令。

mark (文本块标记)

一个辅助性的指针，它与光标一起定义文本块的边界。文本块可以被删除、移动或者复制。GNU Emacs 里的文本块标记是不可见的。

markup (排版标记)

告诉文本排版器程序按某种特定格式打印或显示文本的排版指令代码。**T_EX**和**troff**就是两个能够把内含排版标记的ASCII文件输出为类似于印刷排版效果的文本排版程序的例子。HTML则通过一个能够在屏幕上对它们进行排版的浏览器显示出来。

minibuffer (辅助输入区)

屏幕底部一个供用户输入特定信息的区域。比如说，当你按下“**C-x C-w**”组合键准备保存一个文件时，Emacs将要求在辅助输入区里输入一个文件名。Emacs还会利用这个区域来显示各种提示性的消息。

minor mode (副编辑模式)

独立于主编辑模式而能够被单独启用或禁用的功能。启用自动换行功能的自动换行模式就是一个副编辑模式。

mode (编辑模式)

Emacs用来使其行为适应用户手中工作的定制方式，又分为主编辑模式和副编辑模式。主编辑模式负责定义手里的主要工作，而副编辑模式则是一些可以在主编辑模式里单独启用或禁用的功能。比如说，文本模式就是一种主编辑模式，而启用自动换行功能的自动换行模式就是一个副编辑模式。

mode line (状态行)

Emacs窗口的最后一行，通常呈反显状态或者显示为另外一种颜色。状态行的作用是告诉用户正在编辑的编辑缓冲区、启用的主编辑模式和副编辑模式、在编辑缓冲区里的当前位置等信息。根据具体情况，状态行也可以提供其他信息，比如时间和日期等。

Mosaic

第一个图像界面的Web浏览器，由伊利诺斯大学Urbana-Champaign分校的国家超级计算应用中心开发。很多商业化的浏览器都是以Mosaic为基础的。Netscape就是由许多原来的Mosaic开发者们创建的。

output groups (输出组)

在**shell**模式下，一个“输出组”是由一条**shell**命令和该命令的执行输出组成的。输出组使我们能够更方便地在**shell**模式里的命令之间移动，因为某个给定的命令说不定就有好几页长。

overwrite mode (改写模式)

一个副编辑模式，它允许新输入的字符覆盖（擦除）现有的文本。启用改写模式的命令是“**ESC x overwrite-mode RETURN**”。

paragraph formatting (段落重排)

参见 auto-fill mode (自动换行模式)。

paste (粘贴)

把此前删除或者剪切下来的文本插入到光标当前位置。Emacs在提到粘贴操作的时候使用的是术语“yank”。

pause (等待输入)

一项与宏有关的功能，即临时停下宏的执行等待用户的输入。用户完成输入后再用“**ESC C-c**”组合键继续执行。有关详细讨论请参考第十章。

picture mode (图形模式)

设计目的是允许用户使用键盘字符绘制简单图形的主编辑模式。关于图形模式的详细讨论请参考第八章。

point (插入点)

即光标位置。精确地讲，插入点是位于光标所在字符和前一个字符的中间。在实际工作中，通常可以把光标和插入点看做是同义词。但知道插入点的准确位置能够帮助用户更好地掌握某些命令（比如粘贴或位置交换命令等）。此外，每一个编辑缓冲区都有一个插入点（一个当前位置），而光标则每次只能出现在一个编辑缓冲区里。

query (查询)

一项与宏有关的功能，即由用户决定是否需要继续执行某个宏。

query-replace (查询-替换)

一项与查找与替换操作有关的功能，即由用户逐个地决定是否要进行给定的替换。

read-only (只读)

一个只可以查看不可以修改的文件或编辑缓冲区。但可以把一个只读编辑缓冲区里的文本复制并粘贴到另外一个编辑缓冲区里。

read-write (读写)

一个既可以查看（读）也可以修改（写）的编辑缓冲区。

rectangle editing (矩形编辑)

利用 Emacs 中的矩形概念，能够重新安排和编辑信息的列。关于矩形寄存区的详细讨论请参考第八章。

region (文本块)

光标（或者插入点）与文本块标记之间的区域。可以对文本块进行删除、移动或者复制操作。当前定义的文本块在某些显示器上呈反显状态。如果不是这样，可以按下“**C-x C-x**”组合键（命令名是 **exchange-point-and-mark**）查看文本块的边界。

registers (矩形寄存区)

用来存放矩形或文本的区域。类似于删除环，但为了便于对信息进行（经常会重复进行的）检索，矩形寄存区的名字只能是单个字符。

regular expression (正则表达式)

完成复杂和变化的字符串匹配工作的一项UNIX功能。Emacs支持正则表达式查找操作和正则表达式替换操作。能够用来构造正则表达式的字符清单见第三章；对正则表达式更深入的讨论请参考第十三章。

RMAIL

Emacs的邮件阅读功能组件。

rot13

一个简单的文本加密程序，用来对可能引起别人反感的消息进行加密。它可以被任何人拥有rot13程序的人解密。Gnus能够对rot13文件进行加密和解密操作。

search and replace (查找与替换操作)

参见query-replace（查询—替换）。

search string (查找字符串)

在查找操作中，正在被寻找的文本。

setup file (设置文件)

参见“*.emacs*”文件。

shell buffer (shell 编辑缓冲区)

Emacs创建的一个编辑缓冲区，它里面运行一个UNIX的shell；这样，当与UNIX操作系统互动的时候，就可以利用Emacs的编辑命令和功能，比如它的自动补足功能等。打开一个shell编辑缓冲区的命令是“**ESC x shell**”。

Telnet mode (Telnet模式)

一个面向Telnet的Emacs接口。Telnet是一个网络协议，用户可以通过它登录和使用本地网或因特网上的其他计算机。

uniform resource locator (URL, 统一资源定位器)

通常简称为URL地址，它是某个因特网资源的地址。URL的格式一般为“*protocol://resourcename*”。比如说，WWW资源的URL是“*http://www.ora.com*”形式；FTP档案的URL则是“*ftp://ftp.ora.com*”形式。出现在斜线字符前面的是Web访问指定资源的网络协议（比如FTP）。跟在斜线后面的是被访问的资源的名字。它可以很宽泛（比如O'Reilly & Associates出版公司Web站点的总称*www.ora.com*），也可以很具体，比如像《ora.com杂志》主页的URL：“*http://www.ora.com/www/oracom*”。



variable (变量)

Emacs提供的上百种功能都可以由用户通过设置相应的变量进行定制。如果只是临时性地需要在某次会话对变量进行设置，可以输入“**ESC x set-variable**”命令；如果想做永久性的设置，就需要使用“*.emacs*”文件。关于变量的详细讨论请参考第十一章。

W3

William Perry为Emacs开发的一个WWW浏览器。**W3**并不是Emacs标准发行版本的组件，但它提供了一种把WWW访问功能与Emacs操作集成为一个整体的绝佳方法。**W3**能够与Mosaic共享收藏夹和历史记录文件。

wastebasket (回收站)

参见**kill ring (删除环)**。

window (窗口)

屏幕上显示一个编辑缓冲区的区域。在缺省的情况下，Emacs只有一个窗口。但很多命令，包括帮助命令，都会自动把屏幕拆分为两个窗口。可以拆分出水平窗口（这类窗口最常用）和垂直（呈左右并排）窗口。可以在多个窗口里同时打开同一个编辑缓冲区。**X**用户还可以创建出多个**X**窗口。为了避免概念混乱，多个**X**窗口在Emacs里称为窗格（frame）。

word search (单词查找)

一种忽略换行符和标点符号的查找操作。如果你知道自己的编辑缓冲区里确实存在着某些文本，但用其他查找操作（比如递增查找）却找不到，就可能是其中有一个换行符——递增查找会把换行符解释为一个字符。再用单词查找操作试试。

word wrap (字换行)

参见**auto-fill mode (自动换行模式)**。

World Wide Web (万维网)

通常简称为WWW，这是一种基于超文本的因特网信息服务；通过它，即使用户不了解某些特定因特网应用软件（比如FTP软件）的使用方法，也可以毫无困难地访问很多因特网资源。用户通过选择超链接（hyperlink）漫游WWW。WWW是由Tim Berners-Lee在瑞士日内瓦的European Particle Physics laboratory（CERN，欧洲粒子物理实验室）创立的。

yank (粘贴)

把剪切下来的文本复制到当前编辑缓冲区里。

O'Reilly & Associates 公司介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly & Associates 公司授权机械工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly & Associates 公司是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的 50 本书之一）到 GNN（最早的 Internet 门户和商业网站），再到 WebSite（第一个桌面 PC 的 Web 服务器软件），O'Reilly & Associates 一直处于 Internet 发展的最前沿。

许多书店的反馈表明，O'Reilly & Associates 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly & Associates 公司具有深厚的计算机专业背景，这使得 O'Reilly & Associates 形成了一个非常不同于其他出版商的出版方针。O'Reilly & Associates 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly & Associates 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly & Associates 依靠他们及时地推出图书。因为 O'Reilly & Associates 紧密地与计算机业界联系着，所以 O'Reilly & Associates 知道市场上真正需要什么图书。



《学习 bash》(第二版)

作者: Cameron Newham & Bill Rosenblatt



bash 是自由软件基金会发布的“Bourne Again Shell”的缩写，是流行的 UNIX Bourne shell 的开源软件替代产品，供全球 Linux 用户选用。《学习 bash》(第二版) 正是学习 bash 的权威指南。

本书教授了如何使用 bash 的高级命令行特性，如历史命令、命令行编辑和自动完成命令，还介绍了 shell 编程（这是没有一个 UNIX 或 Linux 用户能够不使用的技巧），阐述了如何使用 bash 的编程特性完成各种功能。读者还将学习到流控制、信号处理、命令行处理和 I/O，及如何调试 bash 程序。

《WEB 设计技术手册》(第二版)

作者: Jennifer Niederst



本书涵盖了在设计 Web 页面的时候需要知道的所有东西。它的编排和组织让读者可以很快地找到问题的答案。这个经过全面修正和扩展的第二版充分展现了各种前沿技术信息，以及 Web 设计人员必须了解的技术信息。

本书是有关 HTML 4.01 标签的优秀参考（包括表格、框架、表单、颜色和级联样式表），同时还包括对浏览器的支持、平台特性和标准的相关信息。读者还可以查阅到很多关于使用图形、多媒体、音频和视频的最新信息，以及诸如动态 HTML、JavaScript 和 XML 等的高级技术。本书还增添了关于 XHTML、WML 和 SMIL 的新章节。本书是各类 Web 设计人员不可缺少的工具。

《lex 与 yacc》(第二版)

作者: John R. Levine, Tony Mason & Doug Brown



《lex 与 yacc》是唯一一本专门介绍这两个重要的 UNIX 编程工具的书。这本新版是完全的修订版，并以很多新的扩充示例代替了旧的示例。几个介绍性章节已经完全重写，还有一章专门介绍实现 SQL 语法，给出了有经验的程序员希望看到的各种细节。

本书对 lex 和 yacc 的重要主题提供了详尽的参考手册。对所有主要的 lex 和 yacc 的 MS-DOS 和 UNIX 版本，本书都进行了介绍，包括 AT&T lex 和 yacc、Berkeley yacc、Berkeley/Gnu flex、Gnu bison、MKS lex 和 Yacc、Abraxas PCYACC、等等。

《ActionScript 权威指南》

作者: Colin Moock



本书是关于 ActionScript 的完整而深入说明，其目标读者既包括初来乍到的 Flash 开发者，又包括那些要将技术转移到 ActionScript 上的 JavaScript 程序员（两种语言都是以 ECMAScript 标准为基础的）。

本书可以让新的 ActionScript 程序员获得迅速的进步。它用传统的形式来说明 ActionScript，给读者打下坚实的理论基础。富有经验的程序员可以在学习 Flash 的复杂部分时利用他们的 JavaScript 知识。在理论之上，本书还包含了很多实际技巧和现实的例子，包括了滚动文本域、菜单按钮、多项选择测试、XML 驱动的站点、物理视频游戏和实际的多用户环境等等。本书还谨慎地讨论了一些未证明或者正在证明的主题。

O'REILLY®
奥莱理软件(北京)有限公司

《JavaScript 权威指南》(第四版)

作者: David Flanagan



本书全面介绍了 JavaScript 语言的核心, 以及 Web 浏览器中实现的遗留和标准的 DOM。它运用了一些复杂的例子, 说明如何处理验证表单数据、使用 cookie、创建可移植的 DHTML 动画等常见任务。本书还包括详细的参考手册, 涵盖了 JavaScript 的核心 API、遗留的客户端 API 和 W3C 标准 DOM API, 记述了这些 API 中的每一个 JavaScript 对象、方法、性质、构造函数、常量和事件处理器。

《JavaScript 权威指南》是 JavaScript 程序设计者的完整指南和参考手册。对于使用最新的、遵守标准的 Web 浏览器 (如 Internet Explorer 6、Netscape 6 和 Mozilla) 的开发者, 它尤其有用。HTML 作者可以从中学习如何用 JavaScript 创建动态网页。经验丰富的程序设计者可以从中学到如何编写复杂的 JavaScript 程序需要的信息。

《Unix 备份与恢复》

作者: W. Curtis Preston



本书完整地覆盖了 Unix 系统备份和恢复领域的方方面面, 并且为各种规模、各种预算的环境提供了实用的、经济的备份和恢复解决方案。本书从介绍 Unix 系统管理员可用的本地备份工具开始, 最后给出了商业备份工具选择的实用建议。对于管理员来说, 本书可以说是无价之宝。

《学习 vi 编辑器》(第六版)

作者: Linda Lamb & Arnold Robbins



vi 是一种配备在大部分 UNIX 系统上的全屏幕文本编辑器。对许多用户来说, 在 UNIX 环境下工作就意味着使用 vi。

这本畅销书的最新修订版是使用 vi 进行文本编辑的完全手册。新增加的内容涵盖了四种 vi 版本: nvi, elvis, vim 和 vile, 以及它们对 vi 的扩展功能, 如多窗口编辑、GUI 接口、扩展的正则表达式和为程序员提供的增强特性等等。还增加了一个描述 vi 在 UNIX 世界和因特网文化中的重要地位的附录。

《UNIX 操作系统》(第五版)

作者: Jerry Peek, Grace Todino & John Strang



本书主要包含以下内容:

- 登录到系统和从系统中注销
- 使用窗口系统
- 管理 UNIX 文件和目录
- 发送和接收邮件
- 阅读新闻组和向新闻组发布消息
- 浏览 Web
- 与朋友和同事交谈
- 理解管道和过滤器
- 后台处理
- 基本网络命令

本书适合 UNIX 初学者使用。

O'REILLY®

奥莱利软件(北京)有限公司

北京海淀区知春路 49 号新中关公寓 B-809 邮政编码: 100080 <http://www.oreilly.com.cn> info@mail.oreilly.com.cn

作者简介

Debra Cameron 是 Cameron 咨询公司的总裁。10多年来，Deb 写了很多与计算机工业有关的书。她写了大量关于局域网、电子商务和因特网安全方面的文章。她的作品曾经获得过技术通信协会（Society for Technical Communication）的表彰。除写作和咨询工作外，她还发表演说和讲授关于如何有效地使用因特网方面的课程。

Deb 毕业于佛罗里达大学。她与她的丈夫 Jim 及两个孩子 Megan 和 David 住在宾夕法尼亚州的 Bellefonte。她喜欢散步、眺望山景和给她的孩子讲故事。她最喜欢的作家包括查尔斯·狄更斯、维克多·雨果、乔治·麦克唐纳、C.S.刘易斯和多萝西·塞亚。

Bill Rosenblatt 居住在费城。他现就职于纽约市 Sun 公司，任企业 IT 体系和数字媒介策略分析师。

他在普林斯顿大学获得 B.S.E 学位，又在位于 Amherst 市的麻萨诸塞大学获得了 M.S. 和 A.B.D 学位，这两个学位都是与计算机科学有关的。他在计算领域的兴趣包括数字图书馆、数字知识产权和因特网/局域网的软件开发工具。在计算以外的领域，他的兴趣集中在法式烹调、古典音乐、爵士乐和福尔摩斯推理探案小说。Bill 与他的妻子 Jessica 居住在曼哈顿的上西区，在他家周围有很多一流的餐馆和书店。

Eric Raymond 是一个自由软件高手，偶尔也写写书。他在各种 Emacs 版本上的程序设计经验差不多有 15 年了。他为 Emacs 19 设计了几个新的功能，包括 VC 模式和 GUD 模式、帮助系统的程序包查找器和程序包上传代码。他还负责几个 FAQ（常见问题答疑）和 Jargon File（行话文件）的维护工作。Eric 住在离费城以西 20 英里远的地方，并在当地帮助运行一个免费因特网的站点，这个站点的名字叫 Chester County Interlink (CCIL)。大家可以在主页 <http://www.ccil.org/~esr/home.html> 上找到更多关于他的自由软件项目和 CCIL 的资料。在他忙里偷闲的自由时间里，人们会看到他在读科幻小说、吹笛子、弹吉他、练空手道、开车兜风，或者与他女朋友的猫一起练心灵感应。



封面介绍

出现在本书封面上的动物是一只非洲角马。这种动物原产非洲，在 Serengeti 平原繁衍生息。雄性非洲角马高度不超过 52 英寸，重量不超过 500 磅，但在同类物种中却有着最具威力的角。成年雄性角马的领土性很强，往往独来独往；而雌性和幼年角马则选择群居生活，在迁徙途中它们往往会展开一个数以千计的大军。非洲角马是狮子最喜欢的一种食物。



图书在版编目 (CIP) 数据

学习 GNU Emacs (第二版) / (美) 卡马伦 (Cameron, D.) 等著; 杨涛等译 .
- 北京: 机械工业出版社, 2002.2

书名原文: Learning GNU Emacs, Second Edition
ISBN 7-111-10348-3

I. 学 ... II. ①卡 ... ②杨 ... III. 文字处理系统, Emacs IV. TP391.12
中国版本图书馆 CIP 数据核字 (2002) 第 039495 号

北京市版权局著作权合同登记
图字: 01-2002-1850 号

Copyright©1996 by O'Reilly & Associates, Inc.

Simplified Chinese Edition, jointly published by O'Reilly & Associates, Inc. and China Machine Press, 2002. Authorized translation of the English edition, 1996 O'Reilly & Associates, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly & Associates, Inc. 出版 1996。

简体中文版由机械工业出版社出版 2002。英文原版的翻译得到 O'Reilly & Associates, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly & Associates, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

书 名 / 学习 GNU Emacs (第二版)
书 号 / ISBN 7-111-10348-3/TP.2457
责任编辑 / 史宗海, 陆颖
封面设计 / Edie Freedman, 张健
出版发行 / 机械工业出版社
地 址 / 北京市西城区百万庄大街 22 号 (邮政编码 100037)
经 销 / 新华书店北京发行所发行
印 刷 / 北京牛山世兴印刷厂
开 本 / 787 毫米 × 1092 毫米 16 开本 37.5 印张 554 千字
版 次 / 2003 年 6 月第一版 2003 年 6 月第一次印刷
印 数 / 0001-4000 册
定 价 / 68.00 元 (册)

(凡购本书, 如有倒页、脱页、缺页, 由本社发行部调换)