本章内容：

- vi的各种克隆版本
- 多窗口编辑
- GUI接口
- 扩展的正则表达式
- 增强的标志
- 改进的功能
- 编程辅助
- 编辑器比较小结
- 后面内容预览

第八章

vi克隆版本的功能总结

vi的各种克隆版本

*vi*编辑器有许多免费可用的“克隆”版本。附录五“vi和Internet”提供了列出所有知名的*vi*克隆版本的Web站点，我们选择了其中最流行的四个进行介绍。它们是：

- Keith Bostic的*nvi* 1.79版
- Steve Kirkendall的*elvis* 2.0版
- Bram Moolenaar的*vim* 5.0版
- Kevin Buettner、Tom Dickey和Paul Fox的*vile* 7.4版

编写这些克隆版本的原因是由于*vi*的源代码不是免费可用的，这使得把*vi*移植到非UNIX环境或研究它的源代码变得不可能，或者是由于UNIX的*vi*（或其他的克隆版本）不能提供想要的功能。例如，UNIX的*vi*通常对行的最大长度有限制并且不能编辑二进制文件（与各种编辑器相关的章节提供了更多有关它们各自的历史信息）。

每个编辑器都对UNIX 的*vi*进行了大量扩展；虽然一些克隆版本提供了相同的扩展，但是它们的扩展方式不同。本章我们不是对在每个编辑器的章节中所讨论的每个普通功能进行重复，而是集中讨论它们的实现方法。你可以把本章认为是对“ 各个克隆版本做什么 ”进行讨论，而每个克隆版本的章节则讨论“ 该版本如何工作 ”。

本章包括下列主题：

多窗口编辑

该功能把屏幕分割为多个“ 窗口 ”（注 1），你可以在每个窗口编辑不同的文件，或者在多个窗口编辑同一个文件。这也许是对常规*vi*的惟一最重要的扩展。

GUI 接口

除*nvi*之外的所有克隆版本都可以将其编译以支持 X Window 接口。如果是在运行 X 的系统中，GUI 版本的使用可能比分割*xterm*（或其他的终端仿真程序）的屏幕效果更好。GUI 版本通常提供像滚动条和多字体那样的优秀功能。也可能支持其他操作系统自带的 GUI。

扩展的正则表达式

所有的克隆版本都使用与UNIX *egrep*（1）命令提供的正则表达式相似或相同的正则表达式进行文本匹配。

增强的标志

正如第七章“ 高级编辑 ”中“ 使用标志 ”一节所介绍的，可以使用*ctags*程序来构建文件的可搜索数据库。通过在进行标志搜索时保存当前位置，克隆版本使标志入栈成为可能。以后还可以返回到那个位置。多个位置可以按后进先出（LIFO）的顺序进行保存，从而形成一个位置栈。

一些*vi*克隆版本的作者和*ctags*克隆版本的作者一起为*ctags*格式的增强版定义了一个标准形式。特别是现在使用允许对函数名重载的C++编写的编辑器，可以使标志功能的使用变得更为容易。

注 1： 要注意这些窗口并不是读者在基于X Window的UNIX工作站上或者在MS-Windows或 Apple Macintosh 下所发现的窗口。

改进的编辑功能

所有的克隆版本都提供了编辑 *ex* 命令行的能力、“ 无限撤消 ”能力、任意长度的行和 8 位数据、增量搜索、（至少一个选项）利用对长行从左到右滚动来取代对长行折行、模式指示以及其他功能。

编程辅助

一些编辑器提供了在典型的“ 编辑－编译－调试 ”软件开发循环期间允许用户保持在编辑器中的功能。

语法高亮显示

在 *elvis*、*vim* 和 *vile* 中，可以使用不同的颜色和 / 或字体显示文件的不同部分。这在编辑程序源代码时尤其有用。

克隆版本中的另外一个功能我们没有进行讨论，那就是扩展语言。在1998年6月，*nvi* 对 Perl 和 Tcl 集成有了基础的支持；*elvis* 有它自己的类似 C 的表达式计算程序（注 2）；*vim* 有类似 C 的表达式计算程序，并且支持 Perl、Python 和 Tcl 集成；一直拥有自己内置扩展语言的 *vile* 支持了 Perl 集成。扩展语言的集成和支持对于所有的编辑器还将继续变化。由于这个原因，任何对扩展语言工具的讨论几乎在该书出版的时候就会变得过时。

如果你对用扩展语言对编辑器进行编程感兴趣，我们建议你检查克隆版本的联机文档（注3）。扩展语言是个值得关注的功能，它们承诺要带给 *vi* 用户全新的感受。由于对用户来说，熟悉 Perl、Python 和 Tcl 那样众所周知的编程语言中的一种或多种是可能的，因此使用它们是很有帮助的。

多窗口编辑

也许各种克隆版本为标准 *vi* 提供的惟一一个最重要的功能就是在多个“ 窗口 ”编

注 2： *elvis* 2.0 文档提到 *elvis* 将拥有真正的扩展语言，它很像 Perl 但是很可能不适用于 2.1 版本。Steve Keikendall 并没有真正把表达式计算程序看做扩展语言。

注 3： 从一开始，*emacs* 用户就一直在做这项工作，这也是它的许多用户对他们的编辑器相当狂热的一个原因。

辑多个文件的能力。这使得同时容易地编辑多个文件成为可能，也使得通过复制和粘贴从一个文件到另一个文件“剪切和粘贴”文本成为可能（注 4）。

支持每个编辑器的多窗口功能的两个基本概念是：缓冲区和窗口。

缓冲区存储将要编辑的文本。这些文本可能来自某个文件，或者可能是最后要写到文件中的新文本。任何给定的文件都只有一个缓冲区与它相关联。

窗口把文件视图放入缓冲区中，这使得你可以在缓冲区中浏览和修改文本。一个缓冲区可能会与多个窗口相关联。在一个窗口中对缓冲区的修改将会反映到在同一缓冲区打开的任何其他窗口中。缓冲区也可能没有与它相关联的窗口，在这种情况下，虽然可以在以后为它打开一个窗口，但是不能使用该缓冲区进行许多处理。关闭在缓冲区上打开的最后一个窗口等效于“隐藏”该文件。如果该缓冲区已改变但是没有将其写到磁盘中，那么编辑器可能会也可能不会让你关闭那个缓冲区打开的最后一个窗口。

当创建新窗口时，编辑器就会分割当前屏幕。对于大部分编辑器，创建的新窗口将显示当前正在编辑的文件的另一个视图。然后切换到想编辑的下一个文件的窗口，并指示编辑器在那里开始编辑该文件。每个编辑器都提供在窗口之间来回切换的 *vi* 和 *ex* 命令以及改变窗口大小、隐藏和重现窗口的功能。

在每个编辑器的章节，我们将展示简单的分割屏幕（编辑同样的两个文件）并介绍如何分割屏幕和在窗口之间移动。

GUI 接口

elvis、*vim* 和 *vile* 还提供能利用鼠标和位图化显示的图形用户接口（GUI）版本。除了支持 UNIX 下的 X Window 外，对于 MS-Windows 或其他窗口化系统的支持也可能是可用的。表 8-1 对不同克隆版本可用的 GUI 进行了总结。

注 4： 在克隆版本中，不需要通过分割屏幕来在文件之间进行复制和粘贴，只有最初的 *vi* 才在切换文件时放弃剪切缓冲区。

表 8-1 可用的 GUI

编辑器	X11	MS-Windows	OS/2	BeOS	Macintosh	Amiga
elvis	3	3	3			
vim	3	3	3	3	3	3
vile	3	3	3			

扩展的正则表达式

可用于 *vi* 搜索和替换正则表达式的元字符已在第六章“ 全局替换 ”中的“ 元字符在搜索模式中的使用 ”一节进行了描述。每个克隆版本把一些形式的扩展正则表达式或者作为选项提供，或者作为一直可用的功能来提供。通常这些正则表达式与 *egrep* 提供的正则表达式相同（或几乎相同）。遗憾的是，每个克隆版本的扩展方式与其他的克隆版本略有不同。

为了使你对扩展的正则表达式的功能有一定了解，我们要在 *nvi* 环境中列举它们。后面每个克隆版本的章节会对该编辑器的扩展语法进行介绍，这里不再对各个例子进行重复。

nvi 扩展正则表达式是由 POSIX 标准定义的扩展正则表达式（ERE）。为了启动该功能，可从 *.nexrc* 文件或 *ex* 冒号提示符下使用 `set extended`。

除了在第六章介绍的标准元字符和在同一章的“ POSIX 的方括号表达式 ”小节提到的 POSIX 括号表达式外，下面的元字符也是可用的：

`|` 指示交替。例如，`a|b` 匹配 *a* 或 *b*。但是，该命令并不只限制于单个字符：`house|home` 匹配字符串 *house* 或 *home*。

`(...)`
用于分组，允许应用额外的正则表达式操作符。例如 `house|home` 可缩写（如果不是简化）为 `ho(use|me)`。`*` 操作符可应用到圆括号上：`(house|home)*` 匹配 *home*、*homehouse*、*househomehousehouse* 等等。

当设置 `extended` 时，使用圆括号分组后的文本产生的效果与在正规 *vi* 中 `\(...)\` 分组后的文本相似，实际匹配的文本可以使用 `\1`、`\2` 等在替换命令的替换部分进行检索。在这种情况下，`\(` 代表实际的左圆括号。

`+` 匹配前面正则表达式的一次或多次出现。这既可以是单个字符，也可以是圆括号中的一组字符。要注意 `+` 和 `*` 之间的区别，`*` 可允许不匹配任何内容，而这里的 `+` 必须至少是一个匹配。例如，`ho(use|me)*` 既匹配 `ho` 也匹配 `home` 和 `house`，但 `ho(use|me)+` 将不匹配 *ho*。

`?` 匹配前面正则表达式的零次或一次出现。这表示出现或者未出现“可选择”的文本。例如，`free?d` 将匹配 *fred* 或 *freed*，除此之外不匹配任何其他文本。

`{...}`

定义区间表达式。区间表达式描述了表示重复次数的计数数字，在下面的描述中，*n* 和 *m* 代表整型常数。

`{`*n*`}`

对前面正则表达式的 *n* 次重复进行匹配。例如，`(home|house){2}` 匹配 *homehome*、*homehouse*、*househome* 和 *househouse*，除此之外不匹配任何其他文本。

`{`*n*`,}`

对前面正则表达式的 *n* 次或更多次的重复进行匹配。可以把它认为“至少 *n* 次”。

`{`*n*`,`*m*`}`

对 *n* 到 *m* 次的重复进行匹配。由于该范围控制着在替换命令中有多少文本将被替换，因此它是很重要的（注 5）。

当没有设置 `extended` 时，*nvi* 使用 `\{` 和 `}\` 来提供同样的功能。

增强的标志

“Exuberant ctags”程序被认为是比 UNIX 的 *ctags* 功能更强的 *ctags* 克隆版本。它

注 5： 虽然 `*`、`+` 和 `?` 操作符使用起来更方便，但是可以将其还原为 `{0,}`、`{1,}` 和 `{0,1}`。

生成了一种扩展的*tags*文件格式，该格式使得标志搜索和匹配变得更灵活和更易于处理。由于 *vi* 的许多克隆版本都支持它，因此我们首先对它进行介绍。

本章还介绍了标志栈：存储用`:tag`或`^]`命令访问过的多个位置的功能。所有克隆版本都提供了标志栈。

Exuberant ctags

“Exuberant ctags”程序是由 Darren Hiebert 编写的，它的主页是 *http://home.hiwaay.net/~darren/ctags*。到编写这本书为止，该程序目前的版本是2.0.3。下面关于该程序的功能列表摘自 *ctags* 发行版中的 *README* 文件。

- 它具有为所有类型的C和C++语言标记生成标志的能力，其中包括类名、宏定义、枚举名枚举值（枚举中的值）、函数（方法）定义、函数（方法）原型/声明、结构成员和类数据成员、结构名、类型定义、联合名和变量。
- 它支持 C 和 C++ 代码。
- 它具有很强的代码解析能力，很容易处理含有`#if`预处理程序的条件指令。
- 它也可以用来打印在源文件中发现的选定对象的可读列表。
- 它支持生成 GNU 的 *emacs*-style 标志文件（*etags*）。
- 它可在 UNIX、QNX、MS-DOS、Windows 95/NT、OS/2 和 Amiga 上运行。一些预编译过的二进制代码可以从 Web 站点上获得。

Exuberant *ctags* 使用下一小节所介绍的格式生成 *tags* 文件。

新的标志格式

传统上，*tags*文件有三个用制表符分开的域：标志名（通常的标识符）、包含标志的源文件和标识符的位置标识。该标识或者是简单的行号，或者是由斜杠或问号包围的`nomagic`搜索模式。而且，*tags* 文件通常是排过序的。

这是由UNIX的*ctags*程序生成的格式。实际上，*vi*的许多版本都允许在搜索模式域中存在任何命令（一个相当大的安全漏洞）。而且，由于未形成文献的实现巧合，如果行以分号和双引号（`;"`）结束，那么这两个字符后面的任何文本都将被忽略（正如双引号在*.exrc*文件中的作用一样，它标志着注释的开始）。

新格式向后兼容传统格式。前面的三个域是相同的：标志、文件名和搜索模式。Exuberant *ctags*只生成搜索模式，而不是任意的命令。扩展属性则放置在分隔符`;"`的后面。各属性之间由制表符隔开，每个属性都由用冒号隔开的两个子域构成。第一个子域描述该属性的关键字，第二个子域是真实值。表8-2列出了所支持的关键字。

表8-2 扩展的ctags的关键字

关键字	含义
kind	值是表示标志的词汇类型的单个字母，它可以是表示函数的`f`、表示变量的`v`等等。由于默认的属性名是`kind`，因此单个字母可以表示标志的类型（如`f`表示函数）
file	表示“静态的”、即文件的局部的标志。值应该是文件的名字
	如果指定为一个空串（如`file:`），应理解为与文件名域相同；加上这种特殊的情况一方面是出于简洁方面的考虑，另一方面是为了提供一种简单的方法来处理那些不在当前目录下的标志文件。文件名域的值总是相对于该*tags*文件本身所在的目录
function	表示局部标志，值是定义这些标志的文件名
struct	表示`struct`中的域，值是该结构的名字
enum	表示`enum`数据类型中的值，值是`enum`类型的名字
class	表示C++成员函数和变量，值为该类的名字
scope	主要的用意是表示C++类成员函数。它通常为`private`，表示私有成员，或者省略以表示默认的公有成员，这样用户可以把标志搜索限制为只对公有成员
arity	表示函数。参数的数量

如果该域不含有冒号，就将其假定为`kind`类型。下面是一些例子：

```
ARRAYMAXED          awk.h       427;" d
AVG_CHAIN_MAX       array.c     38;"  d     file:
array.c             array.c     1;"   F
```

ARRAYMAXED 是在 *awk.h* 中定义的一个 C #define 宏，AVG_CHAIN_MAX 也是一个 C 宏，但是它只是在 *array.c* 中使用。第三行有一些不同：它是一个代表真实源文件的标志，这由带有 -i F 选项的 Exuberant *ctags* 生成，并且允许使用 :tag array.c 命令。更有用的是，可以把光标放在某个文件名上，然后使用 ^] 命令就可以进入该文件。

在每个属性的值域部分，反斜杠、制表符、回车键和换行符应该分别编码为 \\、\t、\r 和 \n。

扩展的 *tags* 文件可能有一些以 !_TAG_ 开头的原始标志。这些标志通常会被排序到文件的前部，它们对识别创建该文件的程序是很有用的。下面是 Exuberant *ctags* 生成的初始标志：

```
!_TAG_FILE_FORMAT       2                /extended format; ..../
!_TAG_FILE_SORTED       1                /0=unsorted, 1=sorted/
!_TAG_PROGRAM_AUTHOR    Darren Hiebert   /darren@hiebert.com/
!_TAG_PROGRAM_NAME      Exuberant Ctags  //
!_TAG_PROGRAM_URL       http://home.hiwaay.net/~darren/ctags   /.../
!_TAG_PROGRAM_VERSION   2.0.3            /with C++ support/
```

编辑器可能利用这些特殊标志来实现特殊功能。例如，*vim* 发现 !_TAG_FILE_SORTED 标志后，如果该文件确实已被排序，它将会使用折半查找而不是线性查找对 *tags* 文件进行搜索。

如果使用 *tags* 文件，我们建议安装 Exuberant *ctags*。

标志栈

ex 的 :tag 命令和 *vi* 的 ^] 模式命令提供了一种根据 *tags* 文件所提供的信息搜索标识符的受限方法。每个克隆版本都通过维护一个标志位置栈而对该功能进行了扩展。每次调用 *ex* 的 :tag 命令，或者使用 *vi* 的 ^] 模式命令，编辑器都会在搜

索指定的标志前保存当前位置。然后就可以使用 ^T 命令或者 *ex* 命令返回到已保存的位置。

下面列举了 Solaris *vi* 的标志栈和一个实例。每个克隆版本处理标志栈的方法将在每个编辑器的各自章节中进行介绍。

Solaris vi

令人惊讶的是，Solaris 2.6版的*vi*支持标志栈。该功能完全没有在Solaris的*ex*(1)和 *vi*(1)手册页中被文档化。出于完整性考虑，我们在表 8-3、表 8-4 和表 8-5 中对 Solaris *vi* 的标志栈进行了总结。Solaris *vi* 中的标志栈相当简单（注 6）。

表 8-3 Solaris vi 的标志命令

命令	功能
`ta[g][!] tagstring`	编辑在 *tags* 文件中定义的包含 *tagstring* 的文件，如果当前缓冲区已修改而且没有保存，! 会强迫 *vi* 切换到新文件
`po[p][!]`	弹出标志栈中的一个单元

表 8-4 Solaris vi 命令模式下的标志命令

命令	功能
`^]`	在 *tags* 文件中寻找光标下的标识符的位置，并移动到那个位置。如果设置标志入栈，当前位置将自动放入标志栈中。
`^T`	返回到标志栈中前面的位置，即弹出一个单元。

表 8-5 Solaris vi 中用于标志管理的选项

选项	功能
`taglength, tl`	控制要搜索的标志中有效字符的数目。默认值 0 表示所有的字符都是有效的

注 6： 该信息是根据经验发现的。YMMV（你的里程数是会变的）。

表 8-5 Solaris vi 中用于标志管理的选项（续）

选项	功能
`tags, tagpath`	该值是要进行搜索标志的文件名列表，默认值是 `"tags /usr/lib/tags"`
`tagstack`	当设置为真（true）时，*vi* 会使每个位置进入标志栈中。使用 `:set notagstack` 来取消标志入栈

为了使你对标志栈的使用有个概念，我们列举了一个使用 Exuberant *ctags* 和 *vim* 的简单例子。

假如你正在编写一个使用 GNU 的 *getopt_long* 函数的程序，并需要了解该函数的更多信息。

GNU 的 *getopt* 由三个文件 *getopt.h*、*getopt.c* 和 *getopt1.c* 组成。

首先创建 *tags* 文件，然后在 *main.c* 中找到主程序后开始对它进行编辑。

```
$ ctags *.[ch]
$ ls
Makefile     getopt.c    getopt.h    getopt1.c   main.c      tags
$ vim main.c
```

击键 **结果**

/getopt

```
/* option processing. ready, set, go! */
for (optopt = 0, old_optind = 1;
  (c = getopt_long(argc, argv, optlist,
                   optab, NULL)) != EOF;
  optopt = 0, old_optind = optind) {
       if (do_posix)
              opterr = TRUE;
```

编辑 *main.c* 并移动到调用 *getopt_long* 的位置。

^]

```
int
getopt_long (argc, argv, options,
             long_options, opt_index)
     int argc;
     char *const *argv;
     const char *options;
     const struct option *long_options;
     int *opt_index;
{
return _getopt_internal (argc, argv, options,
                         long_options,
                         opt_index, 0);
}
"getopt1.c" 189 lines, 4651 characters
```

对 *getopt_long* 进行标志搜索。*vim* 将移动到 *getopt1.c* 上，并把光标放置在 *getopt_long* 的定义上。

这将显示 *getopt_long* 是 *_getopt_internal* 的“ 打包 ”函数，将光标放置在 *_getopt_internal* 上并再次进行标志搜索。

击键 **结果**

8jf_ ^]

```
int
_getopt_internal (argc, argv, optstring,
                  longopts,longind, long_only)
     int argc;
     char *const *argv;
     const char *optstring;
     const struct option *longopts;
     int *longind;
     int long_only;
{
  optarg = NULL;
"getopt.c" 1000 lines, 28705 characters
```

现在已经移动到 *getopt.c* 中，如果要了解 `struct option` 更多的信息，可以把光标移动到 *option* 上再次进行标志搜索。

5jfo; ^]

```
one.) For long options that have a zero 'flag' field, 'getopt'
returns the contents of the 'val' field. */

struct option
{
#if defined (__STDC__) && __STDC__
  const char *name;
#else
  char *name;
#endif
"getopt.h" 133 lines, 4691 characters
```

编辑器移动到 *getopt.h* 中的 `struct option` 定义处。现在可以仔细地查看解释它如何使用的注释了。

```
:tags
  # TO tag            FROM line     in file
  1  1 getopt_long          205     main.c
  2  1 _getopt_internal      75     getopt1.c
  3  1 option                68     getopt.c
```

vim 中的`:tags`命令显示标志栈。

输入三次 `^T` 将使你返回到 *main.c*，这是开始标志搜索的位置。标志功能使你可以很容易地在编辑源文件时到处移动。

改进的功能

四种克隆版本都提供了使简单文本编辑起来更容易、更有效的功能。

编辑 ex 命令行

在输入 *ex* 模式命令时对它们进行编辑的功能，可能包括已保存的 *ex* 历史命令，还有把文件名和命令与选项之类的其他文本补充完整的功能。

没有行长度限制

编辑任意长度的行的功能，还有编辑包含任何 8 位字符的文件的功能。

无限撤消

成功撤消对某个文件进行的所有编辑的功能。

增量搜索

在输入搜索模式的同时搜索文本的功能。

左 / 右滚动

使较长的行延伸到屏幕边缘的后面而不是对其进行换行的功能。

可视模式

选择任意连续的文本块进行某些操作的功能。

模式标识

区别插入模式与命令模式的可见标识，以及当前行号和列号的标识。

命令行历史和完整化

csh、*tcsh*、*ksh*和*bash* 的用户已经知道，重新调用前面的命令、对它们做少量编辑后重新提交使这些 shell 更实用。

相对于shell用户，这种特性对编辑器用户同样有用。遗憾的是，UNIX的*vi* 没有提供任何保存和重新调用 *ex* 命令的功能。

每个克隆版本都对这种缺陷进行了补救。虽然每个编辑器为保存和重新调用历史命令提供了不同的方法，但是每个编辑器所采取的机制都是有效和有用的。

除了命令外，所有编辑器都可以提供某些类型的完整化。例如，你输入了某个文件名的开头，然后输入某个特殊字符（如制表符），编辑器就会把文件名补充完整。所有的编辑器都能把文件名补充完整，某些还可以把其他内容补充完整。细节将在每个编辑器的章节提供。

任意长度的行和二进制数据

这四种克隆版本都可以处理任意长度的行（注7），过去的*vi*版本通常把每行限制在 1000 个字符左右，更长的行将被截断。

这四种克隆版本也是完整8-比特的，意思是它们可以编辑任何包含8位字符的文件，甚至还可以编辑二进制和/或可执行文件。在许多时候，这是非常有用的。你不必告诉每个编辑器文件是二进制的。

nvi

自动处理二进制数据，不需要特殊的命令行或 *ex* 选项。

elvis

在UNIX下，它不区别对待二进制文件与任何其他的文件。在其他的操作系统上，使用*elvis.brf*文件设置 binary选项，以避免换行符的转换问题（*elvis.brf*文件和hex显示模式将在第十章的“ 令人感兴趣的功能 ”节中进行介绍）。

注7： 高达C的 long 类型最大值2 147 483 64。

vim

对行的长度没有限制。当没有设置`binary`时，*vim* 像 *nvi* 一样会自动处理二进制数据。但是，在编辑二进制文件时，需要使用 *—b* 命令行选项或者设置`:set binary`。同时也将设置一些其他的 *vim* 选项，从而可以更容易地编辑二进制文件。

vile

自动处理二进制数据，不需要特殊的命令行或 *ex* 命令选项。

最后，还有一个难处理的细节。传统的 *vi* 总在最后追加换行符来写文件，在编辑二进制文件时，这可能会把一个字符添加到文件中，从而产生问题。默认情况下，*nvi* 和 *vim* 与 *vi* 是兼容的，将会添加那个新行。在 *vim* 中可以通过设置`binary`选项来避免发生这种情况。*elvis* 和 *vile* 从不追加额外的新行。

无限撤消

UNIX 的 *vi* 只允许撤消最后一次的修改，或把当前行恢复到对它进行任何修改之前的状态。所有的克隆版本都提供了“ 无限撤消 ”的功能，可以一直对修改进行撤消，直到撤消到该文件在任何修改之前的状态。

增量搜索

当使用增量搜索时，在你输入搜索模式的时候，编辑器就会进行匹配并在整个文件中移动光标。最后当输入RETURN键时，搜索结束（注 8）。如果以前从没有看过它，那么最初它是相当令人不安的，但是一段时间后就会习惯它了。

elvis 不支持增量搜索，*nvi* 和 *vim* 使用选项启动增量搜索，而 *vile* 使用两个特殊的 *vi* 模式命令。虽然 *vile* 可以在编译时就禁止增量搜索，但是默认情况下增量搜索是打开的。表 8-6 列出了每个编辑器提供的选项。

注 8： *emacs* 一直有增量搜索的功能。

表 8-6 增量搜索

编辑器	选项	命令	行为
nvi	`searchincr`		光标在输入的文件中移动，总是定位到所匹配文本的第一个字符上
vim	`incserach`		光标在输入的文件中移动，*vim* 高亮显示匹配前面输入的模式的文本
vile		`^X S, ^X R`	光标在输入的文件中移动，总是定位到所匹配文本的第一个字符上。`^X S` 增加了对文件进行向前搜索的功能，而 `^X R` 增加了向后搜索的功能

左右滚动

默认情况下，*vi* 和大部分的克隆版本都会使较长的行在屏幕边界处换行。因此，文件的单个逻辑行可能会占用屏幕上多个物理行的空间。

许多时候，如果把长行简单地隐藏在屏幕右边缘的后面可能会比对其进行换行要好。把光标移动到该行上，在向右移动行的同时屏幕也会跟着“ 滚动 ”，该功能在所有的克隆版本中都是可用的。通常，数字选项控制着滚动屏幕的尺寸，布尔选项控制着是换行还是隐藏在屏幕边缘的后面。*vile* 还有把整个屏幕向旁边移动的命令键。表 8-7 列出了每个编辑器中使用水平滚动的方法。

表 8-7 旁边滚动

编辑器	滚动数量	选项	行为
nvi	`sidescroll = 16`	`leftright`	默认情况下为关闭。当启动时，长行只是隐藏在屏幕边缘的后面。每次屏幕向左或向右滚动 16 个字符
elvis	`sidescroll = 8`	`wrap`	默认情况下为关闭。当启动时，长行只是隐藏在屏幕边缘的后面。每次屏幕向左或向右滚动 8 个字符

表 8-7 旁边滚动（续）

编辑器	滚动数量	选项	行为
vim	`sidescroll = 0`	`wrap`	默认情况下为关闭。当启动时，长行只是隐藏在屏幕边缘的后面。当`sidescroll`设置为0时，每次滚动会把光标放置在屏幕的中间；否则，屏幕将按预期数目的字符滚动
vile	`sideways = 0`	`linewrap`	默认情况下为关闭。当启动时，长行将换行。因此，默认设置为把长行隐藏在屏幕边缘的后面。长行的左右边界使用 < 和 > 进行标记。当 `sideways` 设置为0时，每个滚动会移动屏幕的三分之一；否则，屏幕将按预期数目的字符滚动
		`horizscroll`	默认情况下为启动。当启动时，光标沿着长行移动到屏幕的边缘时会替换整个屏幕。当没有启动时，只是当前行被替换；这在速度较慢的显示器上比较有用

vile 有两个额外的命令：`^X ^R` 和 `^X ^L`。这两个命令分别向右和向左滚动屏幕，而不改变光标在该行的当前位置。你不可能移动到光标位置而离开屏幕。

可视模式

通常，*vi* 中的操作作用于像行、单词、字符这样的文本单元或者作用于从当前光标位置到搜索命令所指定的位置之间的文本段。例如，`d/^}`会删除到下面以右大括号开始的行。*elvis*、*vim* 和 *vile* 都提供了明确选择操作要作用的文本区域的机制，尤其是使选择一个矩形文本块然后把操作应用到矩形内的所有文本成为可能。参考每个编辑器的对应章节来了解各种细节内容。

模式标识

vi 有两种模式：命令模式和插入模式。通常，不能通过查看屏幕来识别自己处于

哪种模式。而且，不使用 ^G 或 ex 的 := 命令就知道在文件中的位置通常也是很有用的。

showmode 和 ruler 这两个选项解决了这些问题。这四种克隆版本的选项名字和含义都一样，甚至 Solaris 的 *vi* 也有 showmode 选项。

表 8-8 列出了每个编辑器中的特殊功能。

表 8-8　位置和模式标识符

编辑器	使用 ruler 显示	使用 showmode 显示
nvi	行和列	插入、修改、替换和命令模式标识符
elvis	行和列	输入和命令模式标识符
vim	行和列	插入、替换和可视模式标识符
vile	行、列和文件百分比	插入、替换和覆盖模式标识符
vi	N/A	分别表示打开、输入、插入、追加、修改、替换、替换一个字符和替换模式的分隔模式标识符

elvis 和 *vim* 的 GUI 版本会根据当前模式改变鼠标形状。

编程辅助

vi 主要是作为程序员的编辑器而开发，它有许多功能方便那些编写 C 程序和 *troff* 文档的 UNIX 程序员的工作。（真正的程序员使用 *troff* 编写真正的文档。）一些克隆版本继承了这个传统，并且添加了许多功能，这使得它们对于那些“高级用户”更好用、更有效（注 9）。

其中有两个功能最值得讨论：

注 9：　与所见即所得（WYSIWYG）的观点相比较，UNIX 是所要即所得的操作系统。（为此要感谢 Scott Lee。）

编辑编译加速

elvis、*vim*和*vile*允许你很容易地调用*make*(1)，从编译器中获取错误信息和自动移动到含有错误的行，然后可以修改错误并重新运行 *make*，所有这些都可以在编辑器内进行。

语法高亮显示

elvis、*vim*和*vile*具有在不同类型的文件中高亮显示和/或改变不同语法单元颜色的功能。

编辑编译加速

编程通常由“编译－测试－调试”过程循环组成。修改源代码，编译新代码，然后对它进行测试和调试。在学习新语言时，语法错误尤为常见，因此在编译期间经常停止和重新启动（或挂起和再继续）编辑器是很麻烦的。

elvis、*vim* 和 *vile* 都提供了允许你在编译程序时不离开编辑器的功能。而且，它们还能获取编译器的输出，并使用它自动到达包含错误的每一行（注 10）。始终如一地使用这种功能可以节省时间和提高编程效率。

下面是使用*elvis*的一个例子。由于我们刚开始学习C++，因此从约定性的第一个程序开始：

击键

```
:w hello.c
```

结果

```
#include <stream.h>

void main()
{
        cout << "hello, world!\n ;
}
```

输入程序时忘记了右引号，然后将其写到*hello.c*文件中。

注 10： 从*emacs*用户习惯的功能中继承来的另一个。

:make hello

```
hello.C:5:unterminated string or character
          constant
hello.C:5:possible real start of unterminated
          constant
make: *** [hello] Error 1
Hit <Enter> to continue  _
```

输入执行*make*的:make 命令，进而运行了C++编译器（在该例子中是*g++*）。

:errlist

```
#include <stream.h>

void main()
{
        cout << "hello, world!\n ;
}
line 5: unterminated string or character
         constant
```

:errlist命令把光标移动到含有错误的行并在状态行上显示第一个编译错误信息。

你可以修复错误，重新保存文件，重新运行:make，最终无错误地编译程序。

所有的编辑器都有相似的功能，它们都将补偿文件中的修改，准确地移动到下一个含有错误的行。更多的细节将在每个编辑器的章节提供。

语法高亮显示

elvis、*vim*和*vile*都提供了某种方式的语法高亮显示，它们还提供了语法着色，在具有那种功能的显示器（如在X11或Linux控制台下）上改变文件中不同部分的颜色。参考每个编辑器的章节以获得更多的信息。

编辑器比较小结

大部分编辑器支持上面描述的绝大部分或所有功能。表8-9对每个编辑器所支持的功能进行了总结。当然，该表并不能提供全部的内容，各个细节将在每个编辑器的单独章节中提供。

表 8-9　功能总结表

功能	nvi	elvis	vim	vile
多窗口编辑	3	3	3	3
GUI		3	3	3
扩展的正则表达式	3	3	3	3
增强的标志		3	3	3
标志栈	3	3	3	3
任意长度的行	3	3	3	3
8 位数据	3	3	3	3
无限撤消	3	3	3	3
增量搜索	3		3	3
左右滚动	3	3	3	3
模式标识符	3	3	3	3
可视模式		3	3	3
编辑编译加速		3	3	3
语法高亮显示		3	3	3
多 OS 支持		3	3	3

后面内容预览

后面四章依次介绍 *nvi*、*elvis*、*vim* 和 *vile*，每章都有以下要点：

1. 编辑器的作者和产生原因。
2. 重要的命令行参数。
3. 联机帮助和其他文档。
4. 初始化—程序要读取的文件和环境变量以及读取的次序。
5. 多窗口编辑。
6. GUI 接口。

7. 扩展的正则表达式。
8. 改进的编辑功能（标志栈、无限撤消等）。
9. 编程辅助（编辑编译加速、语法高亮显示）。
10. 该编辑器令人感兴趣的特有功能。
11. 获取源代码的地方以及支持该编辑器的操作系统。

    所有这些版本都用 *gzip*（GNU zip）进行了压缩，如果你没有 *gzip*，可以从 *ftp://ftp.gnu.org/pub/gnu/gzip-1.2.4.tar* 上得到它。对解压缩非UNIX 系统上 *gzip* 生成的 *tar* 文件，*elvis ftp* 站点上现有的 *untar.c* 是一个非常简单的程序。

由于这些编辑器中的每个都在继续发展，因此我们不可能对每个编辑器的功能进行彻底地介绍，有些功能都可能会马上过时。相反，我们只是选择了最重要的部分，并覆盖了那些最可能需要了解而又随着编辑器的发展最不可能改变的功能。如果需要知道如何使用编辑器的每个最新功能，那么应该把每个编辑器的联机文档作为本书的补充说明。