vim 命令(全) 收藏

本章提供足够的信息使你用够使用 Vim 来做基本的编辑。这里提供的方法不一定是高效快捷的。但起码是有效的。花些时间去练习这些命令，这是后面的知识的基础。



下一章：|usr_03.txt| 移动
前一章：|usr_01.txt| 关于本手册
目录：|usr_toc.txt|

---

*02.1*  第一次运行 Vim

启动 Vim 的命令如下：

```
gvim file.txt
```

在 UNIX 操作系统中，你可以在任意命令提示符下输入这个命令。如果你用的是 Microsoft Windows, 启动一个 MS-DOS 窗口，再输入这个命令。

无论哪一种方式，现在 Vim 开始编辑一个名为 file.txt 的文件了。由于这是一个新建
文件，你会得到一个空的窗口。屏幕看起来会象下面这样：

```
+----------------------------------+
|#                                 |
|~                                 |
|~                                 |
|~                                 |
|~                                 |
|"file.txt" [New file]             |
+----------------------------------+
```

('#"是当前光标的位置)

以波纹线(~)开头的行表示该行在文件中不存在。换句话说，如果 Vim 打开的文件不能充满

这个显示的屏幕，它就会显示以波纹线开头的行。在屏幕的底部，有一个消息行指示文件名为 file.txt 并且说明这是一个新建的文件。这行信息是临时的，新的信息可以覆盖它。
VIM 命 令

gvim 命令建立一个新窗口用于编辑。如果你用的是这个命令：

```
vim file.txt
```

则编辑在命令窗口内进行。换句话说，如果你在 xterm 中运行，编辑器使用 xterm 窗口。如果你用的是 Microsoft Window 的 MS-DOS 窗口，编辑器使用这个 MS-DOS 窗口。两个版本显示出来的文本看起来是一样的。但如果你用的是 gvim，就会有其他特性，如菜单条。后面会有更多的描述。

---

*02.2*  插入文本

Vim 是一个多模式的编辑器。就是说，在不同模式下，编辑器的响应是不同的。在普通模式下，你敲入的字符只是命令；而在插入模式，你敲入的字符就成为插入的文本了。
当你刚刚进入 Vim，它处在普通模式。通过敲入"i"命令(i 是 Insert 的缩写)可以启动
插入模式，这样你就可以输入文字了，这些文字将被插入到文件中。不用担心输错了， 你还
能够随后修正它。要输入下文的程序员打油诗，你可以这样敲：

```
iA very intelligent turtle
Found programming UNIX a hurdle
```

输入"turtle"后，你通过输入回车开始一个新行。最后，你输入 <Esc> 键退出插入模式而回到普通模式。现在在你的 Vim 窗口中就有两行文字了：

```
+-----------------------------------+
|A very intelligent turtle          |
|Found programming UNIX a hurdle    |
|~                                  |
|~                                  |
|                                   |
+-----------------------------------+
```

我 在 什 么 模 式?

要看到你在什么模式，输入如下命令：

```
:set showmode
```

你会发现当你敲入冒号后，Vim 把光标移到窗口的最后一行。那里是你输入“冒号命令”(以冒号开头的命令)的地方，敲入回车结束这个命令的输入(所有的冒号命令都用这种方式结束)。

现在，如果你输入 "i" 命令，Vim 会在窗口的底部显示 --INSERT--（中文模式显示的是--插入-- —— 译者注），这表示你在插入模式。

```
+-------------------------------------+
|A very intelligent turtle            |
|Found programming UNIX a hurdle      |
|~                                    |
|~                                    |
|-- INSERT --                         |
+-------------------------------------+
```

如果你输入<Esc>回到普通模式，最后一行又变成空白。

解 决 问 题

Vim 新手常常遇到的一个问题是不知道自己在什么模式下，可能是因为忘了，也可能是因为不小心敲了一个切换模式的命令。无论你在什么模式，要回到普通模式，只要敲 <Esc> 就可以了。有时你需要敲两次，如果 Vim 发出“嘀”一声，就表示你已经在普通模式了。

---

*02.3*  移动光标

回到普通模式后，你可以使用如下命令移动光标：

```
h       左                                              *hjkl*
j       下
k       上
l       右
```

这些命令看起来是随便选的。无论如何，谁听说过用 l 表示右的？但实际上，作这些选择是有理由的：移动光标是最常用的编辑器命令。而这些键位是在你本位的右手。也就是说：这种键位的设定使你可以以最快的速度执行移动操作(特别是当你用十指输入的时候)。

```
Note:
你还可以用方向键移动光标，但这样会减慢你输入的速度，因为你必须把你的手从
```

字母键移动到方向键。想象一下，你在一个小时内可要这样做几百次，这可要花相当多的时间的。

而且，有一些键盘是没有方向键的，或者放在一些很特别的地方。所以，知道 hjkl 的用法在这些情况下就很有帮助了。

记住这些命令的一个方法是：h 在左边，l 在右边，j 指着下面。用图表示如下：

```
    k
h   l
  j
```

学习这些命令的最好方法是使用它。用 "i" 命令输入更多的文字。然后用 hjkl 键移动光标并在某些地方输入一些单词。别忘了用 <Esc> 切换回普通模式。|vimtutor|也是一个练习的好办法。

---

*02.4*  删除字符

要删除一个字符，把光标移到它上面然后输入 "x"。(这是对以前的打字机的一种回归，那时你通过在字符上输入 xxxx 删除它) 例如，把光标移到行首，然后输入 xxxxxxx（七个 x）可以删除 "A very"。结果看起来这样：

```
+-------------------------------------+
|intelligent turtle                   |
|Found programming UNIX a hurdle      |
|~                                    |
|~                                    |
|                                     |
+-------------------------------------+
```

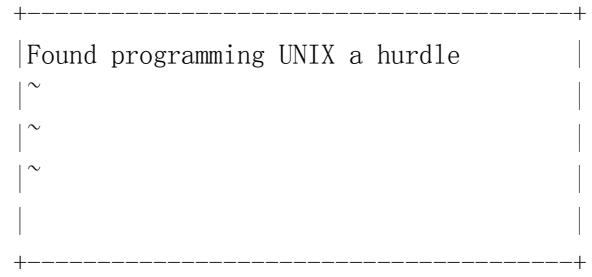
现在你可以输入新的字符了，例如，通过输入：

```
iA young <Esc>
```

这个命令启动一次插入操作(那个"i")，并插入 "A young"，然后退出插入模式(最后一个<Esc>)。结果是：

```
+-------------------------------------+
|A young intelligent turtle           |
|Found programming UNIX a hurdle      |
|~                                    |
|~                                    |
|                                     |
+-------------------------------------+
```

删 除 一 行
要删除一整行，使用 "dd" 命令，后一行会移上来填掉留下的空行：




删 除 一 个 分 行 符

在 Vim 中你可以把两行连起来，这意味着两行间的换行符被删除了。"J" 命令
用于完成这个功能。
以下面两行为例子：

A young intelligent
turtle

把光标移到第一行，然后按 "J"：

A young intelligent turtle

---

*02.5* 撤销与重做

假设现在你删得太多了。当然，你可以重新输入需要的内容。不过，你还有一个更简单的选择。"u" 命令撤销上一个编辑操作。看看下面这个操作：先用 "dd" 删除一行，再敲 "u"，该行又回来了。
再给一个例子：把光标移到第一行的 A 上：

A young intelligent turtle

现在输入 xxxxxxx 删除 "A young"。结果如下：

intelligent turtle

输入"u" 撤销最后一个删除操作。那个删除操作删除字符 g，所以撤销命令恢复这个字符：

g intelligent turtle

下一个 u 命令恢复倒数第二个被删除的字符：

```
ng intelligent turtle
```

下一个 u 命令恢复 u，如此类推：

```
ung intelligent turtle
oung intelligent turtle
young intelligent turtle
young intelligent turtle
A young intelligent turtle
```

Note:
如果你输入 "u" 两次，你的文本恢复原样，那应该是你的 Vim 被配置在 Vi 兼容模式了。看这里修正这个问题：|not-compatible|。
本文假定你工作在 "Vim 的方式"。你可能更喜欢旧的 Vi 的模式，但是你必须小心本文中的一些小区别。

重 做

如果你撤销得太多，你可以输入 CTRL-R（redo）回退前一个命令。换句话说，它撤销一个撤销。要看执行的例子，输入 CTRL-R 两次。字符 A 和它后面的空格就出现了：

```
young intelligent turtle
```

有一个特殊版本的撤销命令："U"（行撤销）。行撤销命令撤销所有在前一个编辑行上的操作。 输入这些命令两次取消前一个 "U"：

```
A very intelligent turtle
  xxxx                  删除 very

A intelligent turtle
              xxxxxx            删除 turtle

A intelligent
                        用 "U" 恢复行
A very intelligent turtle
                        用 "u" 撤销 "U"
A intelligent
```

"U" 命令本身就是一个改变操作，"u" 命令撤销该操作，CTRL-R 命令重做该操作。
有点乱吧， 但不用担心，用 "u" 和 CTRL-R 命令你可以切换到任何你编辑过的状态。

---

*02.6*  其它编辑命令

Vim 有大量的命令可以修改文本。参见|Q_in|和下文。这里是一些经常用到的：

添 加

"i" 命令在光标所在字符前面插入字符。一般情况下，这就够用了，但如果你刚好想在行尾加东西怎么办？要解决这个问题，你需要在文本后插入字符。这通过 "a" (append) 命令实现。

例如，要把如下行

```
and that's not saying much for the turtle.
```

改为

```
and that's not saying much for the turtle!!!
```

把光标移到行尾的句号上。然后输入 "x" 删除它。现在光标处于一行的尾部了，现在输入

```
a!!!<Esc>
```

添加三个感叹号到 turtle 的 "e" 后面：

```
and that's not saying much for the turtle!!!
```

开 始 一 个 新 行

"o" 命令在光标下方建立一个新的空行，并把 Vim 切换到插入模式。然后你可以在这个新行内输入文本了。

假定你的光标在下面两行中第一行的某个地方：

```
A very intelligent turtle
Found programming UNIX a hurdle
```

如果你现在用 "o" 命令并输入新的文字：

```
oThat liked using Vim<Esc>
```

结果会是：

```
A very intelligent turtle
That liked using Vim
Found programming UNIX a hurdle
```

"O" 命令(大写)在光标上方打开一个新行。
指 定 次 数

假定你想向上移动 9 行，你可以输入 "kkkkkkkkk" 或者你可以输入"9k"。实际上，你可以在很多命令前面加一个数字。例如在这章的前面，你通过输入 "a!!!<Esc>" 增加三个感叹号。另一个方法是使用命令 "3a!<Esc>"。次数 3 要求把后面的命令执行三次。同样的，要删除三个字符，可以使用 "3x"。次数总是放在要被处理多次的命令的前面。

---

*02.7*  退出

使用 "ZZ" 命令可以退出。这个命令保存文件并退出。

Note:
与其他编辑器不一样，Vim 不会自动建立一个备份文件。如果你输入 "ZZ"，你的修改立即生效并且不能恢复。你可以配置 Vim 让它产生一个备份文件，参见|07.4|。

放 弃 修 改

有时你会做了一系列的修改才突然发现还不如编辑之前。不用担心，Vim 有“放弃修改并退出”的命令，那就是：

```
:q!
```

别忘了按回车使你的命令生效。

这个命令执行的细节是：命令有三部分，包括冒号(:)，它使 Vim 进入命令模式，q 命令，它告诉 Vim 退出，而感叹号是强制命令修饰符。
这里，强制命令修饰符是必要的，它强制性地要求 Vim 放弃修改并退出。如果你只是输入":q"，Vim 会显示一个错误信息并拒绝退出：

```
E37: No write since last change (use ! to override)
```

通过指定强制执行，你实际上在告诉 Vim:“我知道我所做的看起来很傻，但我长大了，知道自己在做什么。”

如果你放弃修改后还想重新编辑，用 ":e!" 命令可以重新装载原来的文件。

---

*02.8*  寻求帮助

所有你想知道的东西，都可以在 Vim 帮助文件中找到答案，随便问！
要获得一般的帮助用这个命令：

```
:help
```

你还可以用第一个功能键 <F1>。如果你的键盘上有一个 <Help> 键，可能也有效。

如果你不指定主题，":help" 将命令显示一个总揽的帮助窗口。Vim 的作者在帮助系统方面使用了一个很聪明的方案(也许可以说是很懒惰的方案)：他们用一个普通的编辑窗口来显示帮助。你可以在帮助窗口中使用任何普通的 Vim 命令移动光标。所以，h，j，k 和 l 还是表示左，下，上和右。

要退出帮助窗口，用退出一个普通窗口的命令："ZZ"。这只会退出帮助窗口，而不会退出 Vim。

当你阅读帮助的时候，你会发现有一些文字被一对竖线括起来了(例如|help|)。这表示一个超级链接。如果你把光标移到这两个竖线之间并按 CTRL-](标签跳转命令)， 帮助系统会把你引向这个超级链接指向的主题。(由于不是本章的重点，这里不详细讨论，Vim 对超级链接的术语是 "标签"(tag)，所以 CTRL-] 实际是跳转到光标所在单词为名的标签所在的位置。)

跳转几次以后，你可能想回到原来的地方。CTRL-T（标签退栈）把你送回前一个跳转点。CTRL-O (跳转到前一个位置)也能完成相同的功能。

在帮助屏幕的顶上，有一个符号： *help.txt*。这个名字被帮助系统用来定义一个标签
(也就是超级链接的目标)。

参见|29.1|可以了解更多关于标签的内容。

要获得特定主题的帮助，使用如下命令：

```
:help {主题}
```

例如，要获得 "x" 命令的帮助，输入如下命令：

```
:help x
```

要知道如何删除文本，使用如下命令：

```
:help deleting
```

要获得所有命令的帮助索引，使用如下命令：

```
help index
```

如果你需要获得一个包含控制字符的命令的帮助(例如 CTRL-A)，你可以在它前面

加上前缀"CTRL-"。

```
help CTRL-A
```

Vim 有很多模式。在默认情况下，帮助系统显示普通模式的命令。例如，如下命令显示普通模式的 CTRL-H 命令的帮助：

```
:help CTRL-H
```

要表示其他模式，可以使用模式前缀。如果你需要插入模式的命令帮助，使用 "i_" 前缀。例如对于 CTRL-H，你可以用如下命令：

```
:help i_CTRL-H
```

当你启动 Vim，你可以使用一些命令行参数。这些参数以短横线开头(-)。例如知道要-t 这个参数是干什么用的，可以使用这个命令：

```
:help -t
```

Vim 有大量的选项让你定制这个编辑器。如果你要获得选项的帮助，你需要把它括在一个单引号中。例如，要知道 'number' 这个选项干什么的，使用如下命令：

```
:help 'number'
```

如果你看到一个你不能理解的错误信息，例如：

```
E37: No write since last change (use ! to override)
```

你可以使用使用 E 开头的错误号找关于它的帮助：

```
:help E37
```

移 动

在你插入或者删除之前，你需要移动到合适的位置。Vim 有一大堆命令可以移动光标。本章向你介绍最重要的那些。你可以在 |Q_lr| 下面找到这些命令的列表。

|03.9| 简单的查找模式
|03.10| 使用标记

下一章：|usr_04.txt| 做小改动
前一章：|usr_02.txt| Vim 初步
目录：|usr_toc.txt|

---

*03.1* 词移动

要移动光标向前跳一个词，可以使用 "w" 命令。象大多数 Vim 命令一样，你可以在命令前加数字前缀表示把这个命令重复多次。例如，"3w" 表示向前移动 3 个单词。用图表示如下：

```
This is a line with example text
  --->-->->---------------->
   w  w  w    3w
```

要注意的是，如果光标已经在一个单词的词首，"w" 移动到下一个单词的词首。
"b" 命令向后移动到前一个词的词首：

```
This is a line with example text
<----<--<-<---------<---
   b   b b    2b      b
```

还有一个 "e" 命令可以移到下一个单词的词末，而 "ge" 则移动到前一个单词的末尾：

```
This is a line with example text
   <-   <--- ----->   ---->
   ge    ge     e       e
```

如果你在一行的最后一个单词，"w" 命令将把你带到下一行的第一个单词。这样你可以用这个命令在一段中移动，这比使用 "l" 要快得多。"b" 则在反方向完成这个功能。

一个词以非单词字符结尾，例如 ".", "-" 或者 ")"。要改变 Vim 使用的 "分词符"，请参见 'iskeyword' 选项。
还可以以 "空白字符" 分割的 "大单词" 移动。这不是我们通常意义的 "单词"。这就是为什么称之为 "大单词" 的原因。如下图所示：

```
      ge      b          w                          e
      <-     <-         --->                       --->
This is-a line, with special/separated/words (and some more).
   <----- <-----        ------------------>        ----->
     gE      B                  W                    E
      ge      b          w                          e
```

组合运用这些大写和小写的命令，你可以很快地在一段中前后移动。

---

*03.2*  移动到行首或行尾

"$" 命令把光标移动到当前行行尾。如果你的键盘上有 <End> 键，也可以完成相同的功能。

"^" 命令把光标移动到当前行的第一个非空字符，而 "0" 命令则移到一行的第一个字符，<Home> 键也可以完成相同的功能。图示如下：

```
                ^
          <------------
     .....This is a line with example text
     <-----------------   --------------->
             0                   $
```

（这里 "....." 表示空白字符）

象大多数移动命令一样，"$" 命令接受个次数前缀。但是 "移动到一行的行尾 n 次" 没有什么意义，所以它会使光标移动到另一行。例如，"1$" 移动到当前行的行尾，而 "2$" 则移动到下一行的行尾，如此类推。

"0" 命令不能加个数前缀，因为 "0" 本身就是个数字。而且，出人意料地是，"^" 命令也不支持数字前缀。

---

*03.3*  移动到一个指定的字符

单字符查找命令是最有用的移动命令之一。"fx" 命令向前查找本行中的字符 x。提示："f" 表示 "Find"。

例如，假定你在下面例子行的行首，而你想移动到单词 "human" 的 h 那里。只要执行命令 "fh" 即可：

```
     To err is human.  To really foul up you need a computer.
     --------->-------------->
         fh           fy
```

这个例子还演示了用 "fy" 命令移动到 "really" 的词尾。

你可以在这个命令前面加数字前缀，所以，你可以用 "3fl" 命令移动到 "foul" 的 "l":

```
     To err is human.  To really foul up you need a computer.
               ------------------->
                        3fl
```

"F" 命令用于向左查找：

```
        To err is human.  To really foul up you need a computer.
                  <--------------------
                             Fh
```

"tx" 命令与 "fx" 相似，但它只把光标移动到目标字符的前一个字符上。提示："t" 表示 "To"。这个命令的反向版本是 "Tx"。

```
        To err is human.  To really foul up you need a computer.
                  <------------  ------------>
                       Th              tn
```

这四个命令可以通过 ";" 命令重复，"," 命令则用于反向重复。无论用哪个命令，光标永远都不会移出当前行，哪怕是这两行是连续的一个句子。

有时你启动了一个查找命令后才发现自己执行了一个错误的命令。例如，你启动了一个 "f" 命令后才发现你本来想用的是 "F"。要放弃这个查找，输入 <Esc>。所以 "f<Esc>" 取消一个向前查找命令而不做任何操作。 Note: <Esc> 可以中止大部分命令，而不仅仅是查找。

---

*03.4*  括号匹配

当你写程序的时候，你经常会遇到嵌套的()结构。这时，"%" 是一个非常方便的命令了：它能匹配一对括号。如果光标在 "(" 上，它移动到对应的 ")" 上，反之，如果它在 ")" 上，它移动到 "(" 上。

```
                        %
                   <----->
if (a == (b * c) / d)
   <---------------->
                        %
```

这个命令也可适用于 [] 和 {}。(可 用'matchpairs'选项定义)

当光标不在一个有用的字符上，"%" 会先向前找到一个。比如当光标停留在上例中的行首时，"%" 会向前查找到第一个 "("。然后会移动到它的匹配处。

```
if (a == (b * c) / d)
---+--------------->
                %
```

---

*03.5*  移动到指定的行

如果你是一个 C 或者 C++ 程序员，你对下面这样的错误信息应该非常熟悉：

```
prog.c:33: j   undeclared (first use in this function)
```

这表示你可能要移动到 33 行去作一些修改了。那么怎样找到 33 行？一个办法是执行 "9999k" 命令移到文件头，再执行 "32j" 下移到 32 行。这不是一个好办法，但肯定有效。更好的方法是使用 "G" 命令。加上一个次数前缀，这个命令可以把你送到指定的行。例如，"33G" 把你送到 33 行。(要用更好的方法在编译器的错误列表中移动，参见 |usr_30.txt| 的 ":make" 命令部分。)

如果没有数字前缀，"G" 命令把光标移动到文末。移动到文首的命令是 "gg"。"1G" 也能完成这个功能，但稍复杂一点。

```
    |   first line of a file   ^
    |   text text text text     |
    |   text text text text     |  gg
7G  |   text text text text     |
    |   text text text text
    |   text text text text
    V   text text text text     |
        text text text text     |  G
        text text text text     |
        last line of a file     V
```

另一个定位行的方法是使用带数字前缀的 "%" 命令。例如，"50%" 移动到文件的中间，而 "90%" 移到差不多结尾的位置。

前面的描述假定你想移动到文件中的某一行，而不在乎它是否可见。那么如何移动到视野之内的一行呢？下图演示了三个可以使用的命令：

```
          +-------------------------+
H -->     | text sample text        |
          | sample text             |
          | text sample text        |
          | sample text             |
M -->     | text sample text        |
          | sample text             |
          | text sample text        |
          | sample text             |
L -->     | text sample text        |
          +-------------------------+
```

提示："H" 表示 "Home"，"M" 表示 "Middle" 而 "L" 表示 "Last"。

---

*03.6*  确定当前位置

要确定你在文件中的位置，有三种方法：

1.  使用 CTRL-G 命令，你会获得如下消息(假定 'ruler' 选项已经被关闭)：

        "usr_03.txt" line 233 of 650 --35%-- col 45-52

    这里显示了你正在编辑的文件的名称，你所处的当前行的行号，全文的总行数，光标以前的行占全文的百分比，和你所处的列的列号。
    有时你会看到一个分开的两个列号。例如，"col 2-9"。这表示光标处于第二个字符上，但由于使用了制表符，在屏幕上的位置是 9。

2. 设置 'number' 选项。这会在每行的前面加上一个行号：

        :set number

    要重新关闭这个选项：

        :set nonumber

    由于'number'是一个布尔类型的选项，在它前面加上 "no" 表示关闭它。布尔选项只会有两个值，on 或者 off。
    Vim 有很多选项，除了布尔类型的，还有数字或者字符串类型的。在用到的时候会给处一些例子的。

3.  设置 'ruler' 选项。这会在 Vim 窗口的右下角显示当前光标的位置：

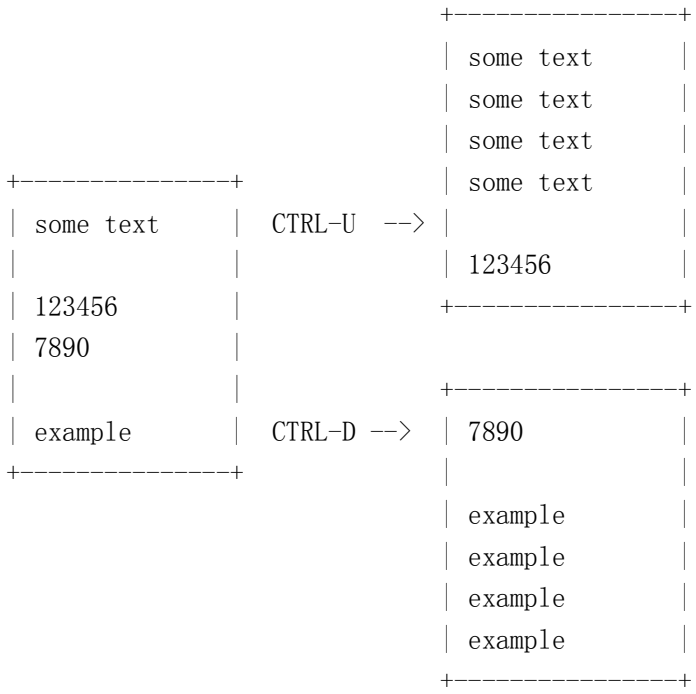
        :set ruler

使用 'ruler' 的好处是它不占多少地方，从而可以留下更多的地方给你的文本。

---
-

*03.7*  滚屏

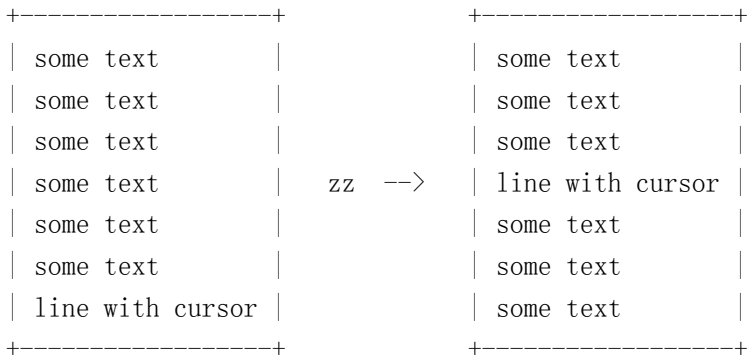
CTRL-U 命令向下滚动半屏。想象一下通过一个视口看着你的文本，然后把这个视口向上移动
该视口的一半高度。这样，窗口移动到当前文字的上面，而文字则移到窗口的下面。不用担心
记不住那边是上。很多人都是这样。
    CTRL-D 命令把视窗向下移动半屏，所以把文字向上移动半屏。




每次滚一行的命令是 CTRL-E（上滚）和 CTRL-Y（下滚）。可以把 CTRL-E 想象为是多给你一行（one line Extra）。

向前滚动一整屏的命令是 CTRL-F（实际上会留下两行不滚动）。反向的命令是 CTRL-B。幸运地，CTRL-F 是向前 (forward) 滚动，CTRL-B 是向后 (backward) 滚动，这比较好记。

移动中的一个常见问题是，当你用 "j" 向下移动的时候，你的光标会处于屏幕的底部，你可能希望，光标所在行处于屏幕的中间。这可以通过 "zz" 命令实现。




"zt" 把光标所在行移动到屏幕的顶部，而 "zb" 则移动到屏幕的底部。Vim 中还有另外一些用于滚动的命令，可以参见|Q_sc|。要使光标上下总保留有几行处于视口中用作上下文，可以使用'scrolloff'选项。

---

*03.8*  简单查找

查找命令是 "/String"。例如，要查找单词 "include"，使用如下命令：

```
/include
```

你会注意到，当你输入 "/" 的时候，光标移到了 Vim 窗口的最后一行，这与 "冒号命令" 一样，在那里你可以输入要查找的字符串。你可以使用退格键 (退格箭头或<BS>) 进行修改，如果需要的时候还可以使用 <Left> 和 <Right> 键。

使用 <Enter> 开始执行这个命令。

> Note:
> 字符.*[]^%/\?~$有特殊含义。如果你要查找它们，需要在前面加上一个 ""。
> 请参见下文。

要查找下一个匹配可以使用 "n" 命令。用下面命令查找光标后的第一个 #include：

```
/#include
```

然后输入 "n" 数次。你会移动到后面每一个 #include。如果你知道你想要的是第几个，可以在这个命令前面增加次数前缀。这样，"3n" 表示移动到第三个匹配点。要注意，"/" 不支持次数前缀。

"?" 命令功能与 "/" 的功能类似，但是是反方向查找：

```
?word
```

"N" 命令在反方向重复前一次查找。因此，在 "/" 命令后执行 "N" 命令是向后查找，在 "?" 命令后执行 "N" 命令是向前查找。

## 忽 略 大 小 写

通常，你必须区分大小写地输入你要查找的内容。但如果你不在乎大小写。可以设置 'ignorecase' 选项：

```
:set ignorecase
```

如果你现在要查找 "word"，它将匹配 "word" 和 "WORD"。如果想再次区分大小写:

```
:set noignorecase
```

## 历 史 记 录

假设你执行了三个查找命令：

```
/one
/two
/three
```

现在，让我们输入 "/" 启动一次查找，但先不按下回车键。现在按 <Up>（上箭头），Vim 把 "/three" 放到你的命令行上。回车就会从当前位置查找 "three"。如果你不回车，继续按 <Up>，Vim 转而显示 "/two"，而下一次 <Up> 变成 "/one"。

你还可以用 <Down> 命令在历史记录中反向查找。

如果你知道前面用过的一个模式以什么开头，而且你想再使用这个模式的话，可以在输入 <Up> 前输入这个开头。继续前面的例子，你可以输入 "/o<Up>"，Vim 就会在命令行上显示 "/one"。

冒号开头的命令也有历史记录。这允许你取回前一个命令并再次执行。这两种历史记录是相互独立的。

## 在文本中查找一个单词

假设你在文本中看到一个单词 "TheLongFunctionName" 而你想找到下一个相同的单词。你可以输入 "/TheLongFunctionName"，但这要输入很多东西。而且如果输错了，Vim 是不可能找到你要找的单词的。

有一个简单的方法：把光标移到那个单词下面使用 "*" 命令。Vim 会取得光标上的单词并把它作为被查找的字符串。

"#" 命令在反向完成相同的功能。你可以在命令前加一个次数："3*" 查找光标下单词第三次出现的地方。

## 查找整个单词

如果你输入 "/the"，你也可能找到 "there"。要找到以 "the" 结尾的单词，可以用：

```
/the>
```

"\>" 是一个特殊的记号，表示只匹配单词末尾。类似地，"\<" 只匹配单词的开头。这样，要匹配一个完整的单词 "the"，只需：

```
/<the>
```

这不会匹配 "there" 或者 "soothe"。注意 "*" 和 "#" 命令也使用了 "词首" 和 "词尾" 标记来匹配整个单词（要部分匹配，使用 "g*" 和 "g#"）

## 高亮匹配

当你编辑一个程序的时候，你看见一个变量叫 "nr"。你想查一下它在哪被用到了。你可以

把光标移到 "nr" 下用 "*" 命令，然后用 n 命令一个个遍历。

这里还有一种办法。输入这个命令：

```
:set hlsearch
```

现在如果你查找 "nr"，Vim 会高亮显示所有匹配的地方。这是一个很好的确定变量在哪被使
用，而不需要输入更多的命令的方法。

要关掉这个功能：

```
:set nohlsearch
```

然后你又需要在下一次查找的时候又切换回来。如果你只是想去掉高亮显示的东西，用如下命令：

```
:nohlsearch
```

这不会复位 hlsearch 选项。它只是关闭高亮显示。当你执行下一次查找的时候，高亮功能会被再次激活。使用 "n" 和 "N" 命令时也一样。

## 调 节 查 找 方 式

有一些选项能改变查找命令的工作方式。其中有几个是最基本的：

```
:set incsearch
```

这个命令使 Vim 在你输入字符串的过程中就显示匹配点。用这个可以检查正确的地方是否已经
被找到了。然后输入 <Enter> 跳到那个地方。或者继续输入更多的字符改变要被查找的字符串。

```
:set nowrapscan
```

这个选项在找到文件结尾后停止查找。或者当你往回查找的时候遇到文件开头停止查找。默认
情况下 'wrapscan' 的状态是 "on"。所以在找到文件尾的时候会自动折返。

## 插 曲

如果你喜欢前面的选项，而且每次用 Vim 都要设置它，那么，你可以把这些命令写到 Vim 的启动文件中。

编辑 |not-compatible| 中提到的文件，或者用如下命令确定这个文件在什么地方：

```
:scriptnames
```

编辑这个文件，例如，象下面这样：

```
:edit ~/.vimrc
```

然后在文中加一行命令来设置这些选项，就好像你在 Vim 中输入一样，例如：

```
Go:set hlsearch<Esc>
```

"G" 移动到文件的结尾，"o" 开始一个新行，然后你在那里输入 ":set" 命令。
最后你用 <Esc> 结束插入模式。然后用如下命令存盘：

```
ZZ
```

现在如果你重新启动 Vim，'hlsearch' 选项就已经被设置了。

---

*03.9*  简单的查找模式

Vim 用正则表达式来定义要查找的对象。正则表达式是一种非常强大和紧凑的定义查找模式的方法。但是非常不幸，这种强大的功能是有代价的，因为使用它需要掌握一些技巧。

本章我们只介绍一些基本的正则表达式。要了解更多的关于查找模式和命令，请参考 |usr_27.txt|。你还可以在|pattern|中找到正则表达式的完整描述。

行 首 与 行 尾

^字符匹配行首。在美式英文键盘上，它在数字键 6 的上面。模式 "include" 匹配一行中任何位置的单词 include。而模式 "^include" 仅匹配在一行开始的 include。

$字符匹配行尾。所以，"was$" 仅匹配在行尾的单词 was。

我们在下面的例子中用 "x" 标记出被 "the" 模式匹配的位置：

```
the solder holding one of the chips melted and the
xxx                      xxx                  xxx
```

用 "/the$" 则匹配如下位置：

```
the solder holding one of the chips melted and the
                                                 xxx
```

而使用 "/^the" 则匹配：

```
the solder holding one of the chips melted and the
xxx
```

你还可以试着用这个模式： "/^the$"，它会匹配仅包括 "the" 的行。并且不包括空格。例如包括 "the " 的行是不会被这个模式匹配的。

匹 配 任 何 单 个 字 符

点 "." 字符匹配任何字符。例如，模式 "c.m" 匹配一个第一个字符是 c，第二个字符是任意字符，而第三个字符是 m 的字符串。例如：

```
We use a computer that became the cummin winter.
         xxx             xxx      xxx
```

匹 配 特 殊 字 符

如果你确实想匹配一个点字符，你可以在前面加一个反斜杠去消除它的特殊含义。

如果你用 "ter." 为模式去查找，你会匹配这些地方：

```
We use a computer that became the cummin winter.
              xxxx                          xxxx
```

但如果你查找 "ter\."，你只会匹配第二个位置。

---

*03.10* 使用标记

当你用 "G" 命令跳到另一个地方，Vim 会记住你从什么地方跳过去的。这个位置成为一个标记，要回到原来的地方，使用如下命令：

```
``
```

` 用单引号 ’ 也可以。

如果再次执行这个命令你会跳回去原来的地方，这是因为 ` 记住了自己跳转前的位置。通常，每次你执行一个会将光标移动到本行之外的命令，这种移动即被称为一个 "跳转"。这包括查找命令 "/" 和 "n"（无论跳转到多远的地方）。但不包括 "fx" 和 "tx" 这些行内查找命令或者 "w" 和 "e" 等词移动命令。

另外 "j" 和 "k" 不会被当做是一次 "跳转"，即使你在前面加上个数前缀也不例外。``命令可以在两个位置上跳来跳去。而 CTRL-O 命令则跳到一个 "较老" 的地方（提示：O 表示 older）。CTRL-I 则跳到一个 "较新" 的地方（提示：I 在键盘上紧靠着 O）。考虑一下如下命令序列：

```
33G
/^The
CTRL-O
```

你首先跳到第 33 行，然后查找以 "The" 开头的一行，然后用 CTRL-O 你会跳回到 33 行。再执行 CTRL-O 你会跳到最初的地方。现在你使用 CTRL-I，就又跳到 33 行。而再用一次 CTRL-I 你又会到达找到 "The" 的地方。

```
      |  example text    ^              |
33G   |  example text    |  CTRL-O      |  CTRL-I
      |  example text    |              |
      V  line 33 text    ^              V
      |  example text    |              |
/^The |  example text    |  CTRL-O      |  CTRL-I
      V  There you are   |              V
         example text
```

Note:
CTRL-I 的功能与<Tab>一样。

":jumps" 命令能输出一个你可以跳往的位置的列表。最后一个你使用的到的标记会用 ">" 符号标记出来。

有 名 字 的 标 记

Vim 允许你在文本中放置自定义的标记。命令 "ma" 用 a 标记当前的光标位置。你可以在文本中使用 26 个标记（a 到 z）。这些标记是不可见的，只是一个由 Vim 记住的位置。

要跳到一个你定义的标记，可以使用命令 `{mark}，这里 {mark} 是指定义标记的那个字母。所以，移到标记 a 的命令是：

```
`a
```

命令 'mark（单引号加上一个标记）会移到标记所在行的行首。这与 `mark 命令是不同的，后者是移到标记所在行上被标记的列。

标记在需要处理一个文件的两个相关地方的时候非常有用。假设你在处理文末的时候需要查看文首的一些内容，先移动到行首，设置一个标记 s (start)：

```
ms
```

然后移动到你需要处理的地方，再设置一个标记 e (end)：

```
me
```

现在你可以随意移动，当你需要看开头的地方，可以使用这个命令移到那里：

```
's
```

然后使用 '' 跳回来。或者用 'e 跳到你正在处理的文尾的地方。
这里使用 s 和 e 作标记名没有特别的含义，只是为了好记而已。

你可以用如下命令取得所有的标记的列表：

:marks

你会注意到有一些特殊的标记，包括：

| | |
|---|---|
| ' | 跳转前的位置 |
| " | 最后编辑的位置 |
| [ | 最后修改的位置的开头 |
| ] | 最后修改的位置的结尾 |

作小改动

本章介绍几种修正和移动文本的方法，这包括三种修改文本的基本方法：操作符－动作，可视模式以及文本对象。




下一章： |usr_05.txt| 选项设置
前一章： |usr_03.txt| 移动
目录： |usr_toc.txt|


---

## *04.1* 操作符与动作

在第二章你已经学过使用 "x" 命令来删除一个字符以及通过个数前缀，例如 "4x" 去删除多个字符。
"dw" 命令删除一个单词。你可能认出来了，"w" 是词移动命令。实际上，"d" 命令后面可以跟任何 "动作"（motion）命令，它会删除从当前位置到光标移动到的目标位置的的全部内容。
例如 "4w" 命令能够向后移动四个单词。所以 "d4w" 命令删除 4 个单词。

```
To err is human. To really foul up you need a computer.
                 ----------------->
                         d4w

To err is human. you need a computer.
```

Vim 只删除从当前位置到 "动作" 把光标移动到的位置的前一个位置。这是因为 Vim 认为你可能不想删掉一个单词的第一个字符。如果你用 "e" 命令作为动作，这时 Vim 认为你是想删掉整个单词（包括最后一个字符）：

```
To err is human. you need a computer.
                --------->
                   d2e

To err is human. a computer.
```

是否包括光标所在的字符取决与你使用的移动命令。在参考手册中，当不包括这个字符时，称为 "非包含的" (exclusive)，而包括这个字符的时候，称为 "包含的" (inclusive)。

"$" 命令移动到行尾。所以，"d$" 命令从当前的位置一直删除到本行行尾。这是一个 "包含的" 命令，所以，这行的最后一个字符也会被删除：

```
To err is human. a computer.
                ------------>
                     d$

To err is human
```

以上定义了一个命令组合模式：操作符－动作。你首先输入一个操作符命令，例如，"d" 就是一个删除操作符。然后你输入一个动作命令，例如 "4l" 或者 "w"。这种方法使你可以在任何你能越过的文本上执行各种操作。

---

*04.2*  修改文本

另一个操作符命令是 "c"，表示修改，change。它的作用方式与 "d" 操作符相似，只是完成后会切换到插入模式。例如，"cw" 修改一个词，更精确的说，它删除一个词，并切换到插入模式。

```
To err is human
   ------->
     c2wbe<Esc>
To be human
```

这里 "c2wbe<Esc>" 包括如下操作：

| | |
|---|---|
| c | 修改操作符 |
| 2w | 移动两个单词的距离(与操作符合起来,它删除两个单词并进入插入模式) |
| be | 插入 be 这个单词 |
| <Esc> | 切换回普通模式 |

如果你留意，你会发现一个奇怪的地方：human 前面的空格没有被删除。有一句谚语说道：任何问题都有一个简单，清楚而错误的回答。"cw" 命令就属于这种情况。c 操作符在很多地方都和 d 一样，但有一个例外，"cw"。它实际上象 "ce" 一样，删除到单词尾。这样单词后面的空格就不包括在内了。这要追溯到使用 Vi 的旧日子。由于很多人已经习惯这种方式了，这个例外就留在 Vim 里了。

## 更 多 的 修 改 命 令

像 "dd" 可以删除一行一样，"cc" 修改一整行。但它会保留这一行的下缩进 (前面的空格)。

"d$" 删除到行尾；"c$" 则修改到行尾。这就象先用 "d$" 删除一行再用 "a" 启动插入模式并加入新的文字。

## 快 捷 键

有些 "操作符－动作" 命令由于经常被使用，所以被设置为单字符命令：

| | | | |
|---|---|---|---|
| x | 表示 | dl | （删除当前光标下的字符） |
| X | 表示 | dh | （删除光标左边的字符） |
| D | 表示 | d$ | （删除到行尾） |
| C | 表示 | c$ | （修改到行尾） |
| s | 表示 | cl | （修改一个字符） |
| S | 表示 | cc | （修改一整行） |

## 在 什 么 地 方 加 入 个 数 前 缀

命令 "3dw" 和 "d3w" 都是删除 3 个单词。如果你非要寻根问底，那么："3dw" 表示删除一个单词 3 次，而 "d3w" 表示删除三个单词一次。这是一个没有分别的分别。实际上你可以放两个次数前缀，例如，"3d2w" 删除两个单词三次，共计六个单词。

## 替 换 单 个 字 符

"r" 命令不是操作符。它只是等你输入一个字符然后用这个字符替换当前光标上的字符。你可以用 "cl" 命令或者 "s" 命令完成相同的功能，但 "r" 命令不需要使用<Esc>

退出插入状态：

```
there is somerhing grong here
rT           rt    rw

There is something wrong here
```

通过个数前缀，"r" 命令可以使多个字符被同一个字符替换，例如：

```
There is something wrong here
                  5rx

There is something xxxxx here
```

要用换行符替换一个字符可以用命令 "r<Enter>"。这会删除一个字符并插入一个换行符。在这里使用个数前缀会删除多个字符但只插入一个换行符："4r<Enter>" 用一个换行符替换四个字符。

---

*04.3*  重复一个修改

"." 是 Vim 中一个非常简单而有用的命令。它重复最后一次的修改操作。例如，假设你在编辑一个 HTML 文件，你想删除所有的 <B> 标记。你把光标移到第一个 "<" 上，然后用 "df>" 命令删除 <B>。然后你就可以移到 </B> 的 < 上面用 "." 命令删除它。"." 命令执行最后一次的修改命令（在本例中，就是 "df>"）。要删除下一个 <B> 标记，移动到下一个 < 的位置，再执行 "." 命令即可。

```
                        To <B>generate</B> a table of <B>contents
f<   找第一个<          --->
df>  删除到>               -->
f<   找下一个<               --------->
.    重复 df>                        --->
f<   找下一个<                          ------------->
.    重复 df>                                       -->
```

"." 命令重复任何除 "u" (undo)，CTRL-R (redo) 和冒号命令外的修改。

再举一个例子：你想把 "four" 修改成"five"。有好几个地方都要作这种修改。你可以用如下命令快速完成这个操作：

```
/four<Enter>    找到第一个 "four"
cwfive<Esc>     修改成 "five"
n               找下一个 "four"
.               重复修改操作
```

```
    n               找下一个 "four"
    .               重复修改
                    如此类推......
```

---

*04.4*  可视模式

要删除一些简单的东西，用 "操作符－动作" 命令可以完成得很好。但很多情况下，并不容易确定用什么命令可以移到你想修改的地方。这时候，你就需要可视模式了。

你可以用 "v" 命令启动可视模式。你可以移动光标到需要的地方。当你这样做的时候，中间的文本会被高亮显示。最后执行一下 "操作符" 命令即可。

例如，要从一个单词的一半删除到下一个单词的一半：

```
This is an examination sample of visual mode
                    ---------->
                      velllld

This is an example of visual mode
```

但你这样做的时候，你不需要真的算要按 l 多少次，你可以在按 "d" 前清楚得看到哪些东西
将要被删除的文本。

如果任何时候你改了注意，只用按一下 <Esc> 就能退出可视模式。

按 行 选 择

如果你想对行做操作，可以使用 "V" 命令来启动可视模式。你会发现在你作任何移动之前，整行都被高亮显示了。左右移动不会有任何效果。而通过上下移动，你可以一次选择多行。

例如，用 "Vjj" 可以选中三行：

```
                    +----------------------+
                    | text more text       |
                 >> | more text more text  | |
selected lines   >> | text text text       | | Vjj
                 >> | text more            | V
                    | more text more       |
                    +----------------------+
```

块 选 择

如果你要处理一个矩形块内的文本，可以使用 CTRL-V 启动可视模式。这在处理表格时非常有用。

| | | | |
|---|---|---|---|
| name | Q1 | Q2 | Q3 |
| pierre | 123 | 455 | 234 |
| john | 0 | 90 | 39 |
| steve | 392 | 63 | 334 |

要删除中间 "Q2" 这一栏，把光标移动到 "Q2" 的 "Q" 上面。按 CTRL-V 启动块可视模式。现在用 "3j" 向下移动三行，然后用 "w" 移到下一个单词。你可以看到最后一栏的第一个字符也被包括进来了。要去掉它，用 "h" 命令即可。现在按 "d"，中间一栏就被删除了。

移 动 到 另 一 端

如果你在可视模式下选中了一些文字，然后你又发现你需要改变被选择的文字的另一端，用 "o" 命令即可（提示："o" 表示 other end），光标会移动到被选中文字的另一端，现在你可以移动光标去改变选中文字的开始点了。再按 "o" 光标还会回到另一端。

当使用块可视模式的时候，你会有四个角，"o" 只是把你移到对角上。而用 "O" 则能移到同一行的另一个角上。

Note: "o" 和 "O" 在可视模式下与在普通模式下的作用有很大的不同；在普通模式下，它们的作用是在光标后或前加入新的一行。

---

*04.5*  移动文本

当你用 "d"，"x" 或者其它命令删除文本的时候，这些文字会被存起来。你可以用 p 命令重新粘贴出来（p 在 Vim 中表示 put）。

看看下面的例子。首先，你会在你要删除的那一行上输入 "dd" 删除一整行，然后移动到你要重新插入这行的地方输入 "p" (put)，这样这一行就会被插入到光标下方。

```
a line          a line        a line
line 2    dd    line 3    p   line 3
line 3                        line 2
```

由于你删除的是一整行，"p" 命令把该行插入到光标下方。如果你删除的是一行的一部分（例如一个单词），"p" 命令会把它插入到光标的后面。

```
Some more boring try text to out commands.
                  ---->
                   dw
```

```
Some more boring text to out commands.
                 -------->
                    welp

Some more boring text to try out commands.
```

关 于 粘 贴 的 更 多 知 识

"P" 命令象 "p" 一样也是插入字符，但插入点在光标前面。当你用 "dd" 删除一行，"P" 会把它插入到光标所在行的前一行。而当你用 "dw" 删除一个单词，"P" 会把它插入到光标前面。

你可以执行这个命令多次，每次会插入相同的文本。

"p" 和 "P" 命令接受个数前缀，被插入的文本就会被插入指定的次数。所以 "dd" 后加一个 "3p" 会把删除行的三个拷贝插入到文本中。

交 换 两 个 字 符

经常发生这样的情况，当你输入字符的时候，你的手指比你的脑子转得快(或者相反？)。这样的结果是你经常把 "the" 敲成 "teh"。Vim 让你可以很容易得修正这种错误。只要把光标移到 "teh" 的 "e" 上，然后执行 "xp" 即可。这个工作过程是："x" 删除一个字符，保存到寄存器。"p" 把这个被保存的字符插入到光标的后面，也就是 "h" 的后面了。

```
teh     th     the
 x       p
```

---

*04.6*  拷贝文本

要把文本从一个地方拷贝到另一个地方，你可以先删除它，然后用 "u" 命令恢复，再用 "p" 拷到另一个地方。这里还有一种简单的办法：抽出 (yank)。"y" 命令可以把文字拷贝到寄存器中。然后用 "p" 命令粘贴到别处。

Yanking 是 Vim 中拷贝命令的名字。由于 "c" 已经被用于表示 change 了，所以拷贝（copy）就不能再用 "c" 了。但 "y" 还是可用的。把这个命令称为 "yanking" 是为了更容易记住 "y" 这个键。(译者注：这里只是把原文译出以作参考，"抽出" 文本毕竟是不妥的。后文中将统一使用 "拷贝"。中文可不存在 change 和 copy 的问题。)

由于 "y" 是一个操作符，所以 "yw" 命令就是拷贝一个单词了。当然了，个数前缀也是有效的。要拷贝两个单词，就可以用 "y2w"。例如：

```
let sqr = LongVariable *
        ------------->
              y2w

let sqr = LongVariable *
                      p

let sqr = LongVariable * LongVariable
```

注意："yw" 命令包括单词后面的空白字符。如果你不想要这个字符，改用 "ye" 命令。

"yy" 命令拷贝一整行，就像 "dd" 删除一整行一样。出乎意料地是，"D" 删除到行尾而 "Y" 却是拷贝一整行。要注意这个区别！"y$" 拷贝到行尾。

```
a text line   yy       a text line           a text line
line 2                 line 2          p     line 2
last line              last line             a text line
                                             last line
```

---

*04.7*  使用剪贴板

如果你使用 Vim 的 GUI 版本 (gvim)，你可以在 "Edit" 菜单中找到 "Copy" 项。你可以先用可视模式选中一些文本，然后使用 Edit/Copy 菜单。现在被选中的文本被拷进了剪贴板。你可以把它粘贴到其它程序，或者在 Vim 内部使用。

如果你已经从其它程序中拷贝了一些文字到剪贴板，你可以在 Vim 中用 Edit/Paste 菜单粘贴进来，这在普通模式和插入模式中都是有效的。如果在可视模式，被选中的文字会被替换掉。

"Cut" 菜单项会在把文字拷进剪贴板前删除它。"Copy"，"Cut" 和 "Paste" 命令在弹出菜单中也有（当然了，前提是有弹出式菜单）。如果你的 Vim 有工具条，在工具条上也能找到这些命令。

如果你用的不是 GUI，或者你根本不喜欢用菜单，你只能用其它办法了。你还是可以用普通的 "y"（yank）和 "p"（put）命令，但在前面必须加上 "*（一个双引号加一个星号）。例如，要拷贝一行到剪贴板中：

```
"*yy
```

要粘贴回来：

```
"*p
```

这仅在支持剪贴板的 Vim 版本中才能工作。关于剪贴板的更多内容请参见 |09.3| 和 |clipboard|。

---

*04.8*  文本对象

如果你在一个单词的中间而又想删掉这个单词，在你用 "dw" 前，你必须先移到这个单词的开始处。这里还有一个更简单的方法："daw"。

```
this is some example text.
              daw

this is some text.
```

"daw" 的 "d" 是删除操作符。"aw" 是一个文本对象。提示："aw" 表示 "A Word"（一个单词），这样，"daw" 就是 "Delete A Word"（删除一个单词）。确切地说，该单词后的空格字符也被删除掉了。

使用文本对象是 Vim 中执行修改的第三种方法。我们已经有 "操作符－动作" 和可视模式两种方法了。现在我们又有了 "操作符－文本对象"。

这种方法与 "操作符－动作" 很相似，但它不是操作于从当前位置到移动目标间的内容，而是对光标所在位置的 "文本对象" 进行操作。文本对象是作为一个整体来处理的。现在光标在对象中的位置无关紧要。

用 "cis" 可以改变一个句子。看下面的句子：

```
Hello there.  This
is an example.  Just
some text.
```

移动到第二行的开始处。现在使用 "cis"：

```
Hello there.    Just
some text.
```

现在你输入新的句子 "Another line."：

```
Hello there.  Another line.  Just
some text.
```

"cis" 包括 "c"（change，修改）操作符和 "is" 文本对象。这表示 "Inner Sentence"（译者注：实在很难用中文表示这个意思了，各位还是记英文名吧）。还有

一个文本对象是 "as"，区别是 "as" 包括句子后面的空白字符而 "is" 不包括。如果你要删除一个句子，而且你还想同时删除句子后面空白字符，就用 "das"；如果你想保留空白字符而替换一个句子,则使用"cis"。

你还可以在可视模式下使用文本对象。这样会选中一个文本对象，而且继续留可视模式，你可以继续多次执行文本对象命令。例如，先用 "v" 启动可视模式，再用 "as" 就可以选中一个句子。现在重复执行 "as"，就会继续选中更多的句子。最后你可以使用一个操作符去处理这些被选中的句子。

你可以在这里找到一个详细的文本对象的列表：|text-objects|。

---

*04.9*  替换模式

"R" 命令启动替换模式。在这个模式下，你输入的每个字符都会覆盖当前光标上的字符。这会一直持续下去，直到你输入<Esc>。

在下面的例子中，你在 "text" 的第一个 "t" 上启动替换模式：

```
This is text.
        Rinteresting.<Esc>

This is interesting.
```

你可能会注意到，这是用十二个字符替换一行中的五个字符。如果超出行的范围，"R" 命令自动进行行扩展，而不是替换到下一行。

你可以通过 <Insert> 在插入模式和替换模式间切换。

但当你使用 <BS>（退格键）进行修正时，你会发现原来被替换的字符又回来了。这就好像一个 "撤消" 命令一样。

---

*04.10* 结论

操作符，移动命令和文本对象可以有各种组合。现在你已经知道它是怎么工作了，你可以用 N 个操作符加上 M 个移动命令，组合出 N*M 个命令！

你可以在这里找到一个操作符的列表：|operator|

还有很多方法可以删除文本。这是一些经常用到的：

| | |
|---|---|
| x | 删除光标下的字符（"dl"的缩写） |
| X | 删除光标前的字符（"dh"的缩写） |
| D | 从当前位置删除到行尾（"d$"的缩写） |

dw　　从当前位置删除到下一个单词开头
db　　从当前位置删除到前一个单词的开头
diw　　删除光标上的单词（不包括空白字符）
daw　　删除光标上的单词（包括空白字符）
dG　　删除到文末
dgg　　删除到文首

如果你用 "c" 代替 "d"，这会变成修改命令；而改用 "y"，则变成拷贝命令，等等等等。

还有一些常用的命令，放在哪一章都不合适，列在这里：

~　　修改光标下字符的大小写，并移动到下一个字符。这不是一个操作符（除非设置了 'tildeop'），所以你不能连接一个动作命令。这个命令在可视模式下也有效，它会改变被选中的所有文本的大小写。

I　　移到当前行的第一个非空字符并启动插入模式

A　　移动到行尾并启动插入模式

Vim 可以按你的需要进行设置。本章告诉你怎样使 Vim 用你指定的选项启动，怎样增加插件
以增强 Vim 的功能；以及怎样进行宏定义。



下一章：|usr_06.txt|　使用语法加亮
前一章：|usr_04.txt|　做小改动
目录：|usr_toc.txt|

---

*05.1*　vimrc 文件

可能你已经厌倦了输入那些经常用到的命令了。要让 Vim 用你习惯的设置启动，你可以把这些设置写到一个叫 vimrc 的文件中。Vim 会在启动的时候读入这个文件。

如果你不知道你的 vimrc 在什么地方，可以使用如下命令：

:scriptnames

命令列出的文件列表开头的几个中应该有一个叫 ".vimrc" 或者 "_vimrc" 的文件在你的 home 目录中。

如果你还没有 vimrc，请参考 |vimrc| 一节看看你应该在什么地方创建 vimrc 文件。另外 ":version" 命令能告诉你 vim 在什么地方找 "用户的 vimrc 文件"。

对于 Unix 系统，肯定是如下文件：

```
~/.vimrc
```

对于 MS-DOS 和 MS-Windows，常常使用下面其中一个文件：

```
$HOME/_vimrc
$VIM/_vimrc
```

vimrc 文件可以包含任何冒号命令。最简单的是设置选项命令。例如，如果你想 Vim 启动的时候始终开启 'incsearch' 选项，可以在你的 vimrc 文件中加上：

```
set incsearch
```

要使这个命令生效，你需要重启动 Vim。后面我们还会学到如何不退出 Vim 就能让它生效。

这一章只解释最基本的东西。要学习更多关于写 Vim 脚本的知识，请参见|usr_41.txt|。

---

*05.2*  vimrc 示例解释

在第一章中，我们曾经介绍过怎样用 vimrc 示例文件（包括在 Vim 发布中）使 Vim 以非 vi 兼容模式启动（参见|not-compatible|）。这个文件可以在这里找到：

```
$VIMRUNTIME/vimrc_example.vim
```

我们在这一节中介绍这个文件中用到的一些命令。这会对你自行参数设置有一定的帮助。但我们不会介绍所有的内容。你需要用 ":help" 获得更多的帮助。

```
set nocompatible
```

如第一章所述，这个命令人为地告诉 Vim 工作在 vi 增强模式，因此与 Vi 不完全兼容。要关闭 'compatible' 选项设，'nocompatible' 可以用于完成这个功能。

```
set backspace=indent,eol,start
```

这指明在插入模式下 <BS> 如何删除光标前面的字符。逗号分隔的三个值分别指：行首的空白字符，分行符和插入模式开始处之前的字符。

```
set autoindent
```

这使 Vim 在启动一个新行的时候使用与前一行一样的缩进。就是说，新行前面会有同样多的空白字符。启动新行是指用 <Enter> 换行，在普通模式下执行 "o" 命令等情况。

```
if has("vms")
  set nobackup
else
  set backup
endif
```

这告诉 Vim 当覆盖一个文件的时候保留一个备份。但 VMS 系统除外，因为 VMS 系统会自动产生备份文件。备份文件的名称是在原来的文件名上加上 "~" 字符。参见 |07.4|

```
set history=50
```

这个命令保存 50 个命令和 50 个查找模式的历史。如果你想 Vim 记住多些或者少些命令，可以把这个数改成其它值。

```
set ruler
```

总在 Vim 窗口的右下角显示当前光标位置。

```
set showcmd
```

在 Vim 窗口右下角，标尺的右边显示未完成的命令。例如，当你输入 "2f"，Vim 在等你输入要查找的字符并且显示 "2f"。当你再输入 w，"2fw" 命令被执行，"2f" 自动消失。

```
+---------------------------------------------+
|text in the Vim window                       |
|~                                            |
|~                                            |
|-- VISUAL --                  2f     43,8   17% |
+---------------------------------------------+
 ^^^^^^^^^^                   ^^^^^^^ ^^^^^^^^^^

  'showmode'                 'showcmd'  'ruler'
```

```
set incsearch
```

在查找模式输入完前显示匹配点。

```
map Q gq
```

这定义一个键映射。下一节会介绍更多的相关内容。这将定义 "Q" 命令用来完成与 "gq" 操作符相同的功能，这是在 Vim 5.0 版前 "Q" 命令的作用。如果没有设置这个映射，"Q" 会启动 Ex 模式，这也许不是你想要的情况。

```
vnoremap p <Esc>:let current_reg = @"<CR>gvs<C-R>=current_reg<CR><Esc>
```

这是一个复杂映射。这里不介绍它是怎么工作的。它的作用是使 "p" 命令在可视模式下用拷贝的字符覆盖被选中的字符。你可以看到映射可以被用来执行相当复杂的操作。但其本质依然是一个命令序列而已，与你直接输入没有什么两样。

```
if &t_Co > 2 || has("gui_running")
  syntax on
  set hlsearch
endif
```

这激活语法加亮功能，但仅在颜色功能有效的时候才有效。而 'hlsearch' 选项告诉 Vim 加亮上次查找模式匹配的地方。"if" 命令在设置选项的时候非常有用，它使设置命令在某些条件下才执行。更多的内容请参见|usr_41.txt|。

```
*vimrc-filetype*
    filetype plugin indent on
```

这启动三个非常灵巧的机制：

1. 文件类型探测
   当你开始编辑一个文件的时候，Vim 会试图确定这个文件的类型。当你编辑 "main.c" 时，Vim 会根据扩展名 ".c" 认为这是一个 C 源文件。当你编辑一个文件前面是 "#!/bin/sh" 的文件时，Vim 会把它认作 "sh" 文件。文件类型探测用于语法加亮和以下另两项。请参见|filetypes|。

2. 使用文件类型相关的插件
   不同的文件需要不同的选项支持。例如，当你编辑一个 "c" 文件，用 'cindent' 选项来自动缩进就非常有用。这些文件类型相关的选项在 Vim 中是通过文件类型插件来实现的。你也可以加入自己的插件，请参见 |write-filetype-plugin|。

3. 使用缩进文件

当编辑程序的时候，行缩进通常可以被自动决定。Vim 用不同的策略处理不同的文件类型。请参见 |:filetype-indent-on| 和 'indentexpr'。

```
autocmd FileType text setlocal textwidth=78
```

这使 Vim 在一行长于 78 个字符的时候自动换行，但仅对纯文本文件中有效。这里包括两个部分。其中 "autocmd FileType text" 定义个自动命令，表示当文件类型被设置为 "text" 的时候，后面的命令自动执行。"setlocal textwidth=78" 设置 'textwidth' 选项为 78，但仅对本文件有效。

```
autocmd BufReadPost *
     if line("'\"") > 0 && line("'\"") <= line("$") |
       exe "normal g`\"" |
     endif
```

这是又一个自动命令。这回它设置为在读入任何文件之后自动执行。后面那堆复杂的东西检查 '" 标记是否已被定义，如果是，则跳转到该标记。行首的反斜杠用于把所有语句连接成一行。这可以避免一行写得太长，请参见 |line-continuation|。这只在 Vim 脚本文件中有效，在命令行中无效。

---

*05.3*  简单键盘映射

映射可以使你把一系列 Vim 命令绑定为一个单键。假设你要用一个大括号将一个特定的单词括起来。例如，把 "amount" 变成 "{amount}"。用 ":map" 命令，就可以让 F5 来完成这个工作。命令如下：

```
:map <F5> i{<Esc>ea}<Esc>
```

Note:
在输入这个命令时，<F5> 要用四个字符表示。相似地，输入 <Esc> 不是直接按 <Esc> 键，而是输入五个字符。在读这份手册要注意这些区别！

让我们来分解一下这个命令：

| | |
|---|---|
| <F5> | F5 功能键。这是命令的触发器。当这个键被按下时，相应的命令即被执行。 |
| i{<Esc> | 插入 { 字符。<Esc> 键用于退出插入模式。 |
| e | 移动到词尾。 |
| a}<Esc> | 插入 } 到单词尾。 |

执行 ":map" 命令后，要在单词两端加上 {}，只需要移到单词上的第一个字符并按 F5。

在这个例子中，触发器是一个单键；它还可以是任何字符串。但若你使用一个已经存在的 Vim 命令，该命令将不在有效。所以你最好避免出现这种情况。

一个可用于映射的键是反斜杠。因为你很可能想定义多个映射，那就加上另一个字符。你可以映射 "\p" 为在单词两端加园括号，而映射 "\c" 为加花括号，例如：

```
:map \p i(<Esc>ea)<Esc>
:map \c i{<Esc>ea}<Esc>
```

你需要在敲入\后，立即敲入 p，以便 Vim 知道它们组成一个命令。

":map" 命令（无参数）列出当前已定义的映射，至少会包括普通模式下的那些。更多的内容参见 |40.1|。

---

*05.4*  增加插件 *add-plugin* *plugin*

Vim 可以通过插件增强功能。插件其实是一个当 Vim 启动的时候能被自动执行的脚本。简单地把插件放到你 Vim 的 plugin 目录中就可以使它生效。
（这个功能在 |+eval| 被编译进 Vim 中时才有效）

Vim 中有两种插件：
全局插件：用于所有类型的文件
文件类型插件：仅用于特定类型的文件
我们将先讨论全局插件，然后涉及文件类型插件 |add-filetype-plugin|。

全局插件 *standard-plugin*

当你启动 Vim，它会自动加载一些插件。你不需要为此做任何事。这些插件增加一些很多人想用的，但由 Vim 脚本实现而非编译进 Vim 中的功能。你可以在帮助索引中找到这些插件：|standard-plugin-list|。还可以参照 |load-plugin|。

*add-global-plugin*
你可以加入一个全局插件使得某些功能在你每次使用 Vim 时都被开启。添加一个全局插件只要两步：
1. 获得一个插件的拷贝
2. 把它塞进合适的目录

获得一个全局插件

在什么地方可以找到插件？
- 有一些与 Vim 一起发布，你可以在 $VIMRUNTIME/macros 目录或其子目录中找到。
- 从网上下载，查一下这个地方：[url]http://vim.sf.net。[/url]
- 在 Vim 的邮件列表中找：|maillist|。
- 自己写一个，参见|write-plugin|。

使 用 一 个 全 局 插 件

首先阅读插件包括的说明文字，看看有没有什么特殊的限制。然后拷贝到你的插件目录：

| 系统 | 插件目录 |
| --- | --- |
| Unix | ~/.vim/plugin/ |
| PC and OS/2 | $HOME/vimfiles/plugin 或 $VIM/vimfiles/plugin |
| Amiga | s:vimfiles/plugin |
| Macintosh | $VIM:vimfiles:plugin |
| RISC-OS | Choices:vimfiles.plugin |

以 Unix 系统为例（假设你还没有 plugin 目录 ）：

```
mkdir ~/.vim
mkdir ~/.vim/plugin
cp /usr/local/share/vim/vim60/macros/justify.vim ~/.vim/plugin
```

就是这样了！现在你可以用这个插件定义的命令调整文字了。

文件类型插件 *add-filetype-plugin* *ftplugins*

Vim 的发布中包括一套针对不同文件类型的插件。你可以用如下命令启用它们：

```
:filetype plugin on
```

这样就行了！ 参阅 |vimrc-filetype|。

如果你缺少某种文件类型的插件，或者你找到一个更好的，你可以自行添加一个。这也只需两步：
1. 获取一个插件的拷贝
2. 塞到合适的目录。

取 得 文 件 类 型 插 件

你可以在找全局插件的相同地方找到文件类型插件。注意一下插件有没有注明文件类型，据此你可以知道这个插件是全局的还是文件类型相关的。在 $VIMRUNTIME/macros 中的是全局插件；文件类型插件在 $VIMRUNTIME/ftplugin 中。

## 使 用 文 件 类 型 插 件 *ftplugin-name*

你可以通过把插件文件放到合适的目录中来增加一个插件。目录的名字与前面提过的全局插件的位置一样，但最后一级目录是 "ftplugin"。假设你找到一个用于 "stuff" 文件类型的插件，而且你的系统是 Unix。那么，你可以把这个文件用如下命令移入 ftplugin 目录：

```
mv thefile ~/.vim/ftplugin/stuff.vim
```

如果这个文件已经存在，你可以检查一下两个插件有没有冲突。如果没有，你可以用另一个名字：

```
mv thefile ~/.vim/ftplugin/stuff_too.vim
```

这里，下划线用来分开文件类型和其它部分（这些部分可以由任意字符组成）。但如果你用 "otherstuff.vim" 就不行了。那是用于 "otherstuff" 类型的文件的。

在 MS-DOS 中不能使用长文件名。如果你增加第二个插件，而这个插件超过 6 个字符，你就没法用了。你可以通过使用另一层目录来解决这个问题：

```
mkdir $VIM/vimfiles/ftplugin/fortran
copy thefile $VIM/vimfiles/ftplugin/fortran/too.vim
```

总的来说，一个文件类型相关的插件的名称是：

```
ftplugin/<filetype>.vim
ftplugin/<filetype>_<name>.vim
ftplugin/<filetype>/<name>.vim
```

这里 "<name>" 可以是任何你喜欢的名字。例如，在 Unix 下，"stuff" 文件类型的插件可以是：

```
~/.vim/ftplugin/stuff.vim
~/.vim/ftplugin/stuff_def.vim
~/.vim/ftplugin/stuff/header.vim
```

这里，<filetype> 部分是相应文件类型的名称。只有对应文件类型的文件才会用这个插件内的设置。插件的 <name> 部分则不重要，你可以对同一个文件类型使用多个插件。Note 插件必须以 ".vim" 结尾。

---

*05.5*  增加帮助　　*add-local-help*

如果幸运的话，你安装的插件还会包括帮助文件。我们这里解释如何安装这个帮助文件，以便你能方便地获得新插件的帮助。

我们以 "matchit.vim" 插件为例（包括在 Vim 中）。这个插件使 "%" 命令可以在两个对应的 HTML 标记间，Vim 脚本的 if/else/endif 间等匹配点间跳转。这非常有用，但它不向前兼容（这也是为什么默认的情况下它不会被激活）。

这个插件有一个文档："matchit.txt"。我们先来把该插件拷贝到合适的位置。这次，我们在 Vim 内完成这个工作，以便利用$VIMRUNTIME。（如果某些目录已经存在你可以省略一些 "mkdir" 命令）

```
:!mkdir ~/.vim
:!mkdir ~/.vim/plugin
:!cp $VIMRUNTIME/macros/matchit.vim ~/.vim/plugin
```

现在在某个 'runtimepath' 目录中建立一个 doc 目录。

```
:!mkdir ~/.vim/doc
```

再把帮助文件拷贝进去：

```
:!cp $VIMRUNTIME/macros/matchit.txt ~/.vim/doc
```

现在开始玩技巧了，怎样使 Vim 允许你跳转到新的主题上？用 |:helptags| 命令产生一个本地的 tags 文件即可：

```
:helptags ~/.vim/doc
```

现在，你可以用这个命令

```
:help g%
```

来获得 "g%" 的帮助了。在使用如下命令的时候，可以看见一个新的条目：

```
:help local-additions
```

本地帮助的标题行被自动的加入到该节了。在那里你可以看到 Vim 添加了那些本地的帮助文件。你还可以从这里跳转到新的帮助中。

要写一个本地帮助文件，请参考 |write-local-help|。

---

*05.6*  选项窗口

如果要找一个选项，你可以查找这个位置的帮助：|options|。另一个方法是用如下命令：

```
:options
```

这会打开一个新窗口，其中给出一个选项的列表，并对每个选项提供一行解释。这些选项被根据种类分组。把光标移到一个主题上然后按回车就可以跳转到那里。再按一下回车或者 CTRL-O 就可以跳回来。

你可以通过这个窗口改变一个选项的值。例如，移到 "displaying text" 主题。然后把光标下移到这一行：

```
set wrap        nowrap
```

当你在上面键入回车，这行会改变为：

```
set nowrap      wrap
```

现在，这个选项被关闭了。

这行的上面是对这个选项的简要描述。将光标向上移动一行，然后按 <Enter>，你可以跳转到 'wrap' 的完整帮助，再用 CTRL-O 可以跳回来。

对于那些值为数字或者字符串的选项，你可以编辑它的值，然后按 <Enter> 来启用该值。例如，把光标移动到下面这行：

```
set so=0
```

用 $ 移到行尾，再用 "r5" 命令修改为五，然后按 <Enter> 使修改生效。现在如果你移动一下光标，你会发现在你的光标移到窗口边界前，你的文字就开始滚动了。这就是选项 'scrolloff' 完成的功能：它指定在距离边界多远的地方开始滚动文字。

---

*05.7*  常用选项

Vim 中有很多选项。大部分你很少用得上。我们在这个介绍一些常用的。别忘了你可以通过 ":help" 命令获得更多的帮助。方法是在选项命令前后加上单引号，例如：

```
:help 'wrap'
```

如果你搞乱了一个选项，你可以通过在选项后加上一个 & 号把它恢复到默认值。例如：

```
:set iskeyword&
```

## 禁 止 折 行

Vim 通常会对长行自动换行，以便你可以看见所有的文字。但有时最好还是能让文字在一行中显示完。这样，你需要左右移动才能看到一整行。以下命令可以切换换行方式：

```
:set nowrap
```

当你移到到那些不能显示的文字上，Vim 会自动向右滚动让你看到后面的文字，要一次滚动十个字符，这样就行了：

```
:set sidescroll=10
```

这个命令不改变文件中的文字，只改变显示方式。

## 移 动 命 令 换 行

很多命令只能在一行中移动。你可以通过 'whichwrap' 选项改变它。如下命令把这个选项设为默认值：

```
:set whichwrap=b,s
```

这样，当光标处于行首时用 <BS> 键可以回到前一行的结尾；当处于行尾时用 <Space> 键可以移动到下一行的行首。

要允许 <Left> 和 <Right> 键也能这样，可以用这个命令：

```
set whichwrap=b,s,<,>
```

这只在普通模式中有效，要在插入模式中也有效，可以：

```
:set whichwrap=b,s,<,>,[,]
```

还有一些可以用的标志，参见 'whichwrap'。

## 显 示 TAB 键

文件中有 TAB 键的时候，你是看不见的。要把它显示出来：

```
:set list
```

现在 TAB 键显示为 ^I，而 $显示在每行的结尾，以便你能找到可能会被你忽略的空白字符在哪里。

这样做的一个缺点是在有很多 TAB 的时候看起来很丑。如果你使用一个有颜色的

终端，或者使用 GUI 模式，Vim 可以用高亮显示空格和 TAB。 使用 'listchars' 选项：

```
:set listchars=tab:>-,trail:-
```

现在，TAB 会被显示成 ">---" 而行尾多余的空白字符显示成 "-"。看起来好多了，是吧？

## 关 键 字

'iskeyword' 选项指定那些字母可以出现在一个单词中：

```
:set iskeyword
<        iskeyword=@,48-57,_,192-255
```

"@" 表示所有字母。"48-57" 表示 ASCII 字符 48-57 ，即数字 0 到 9。"192-255" 是可打印的拉丁字符。

有时你希望横线也是关键字，以便 "w" 命令会把 "upper-case" 看作是一个单词。你可以这样做：

```
:set iskeyword+=-
:set iskeyword
iskeyword=@,48-57,_,192-255,-
```

看一下新的值，你会发现 Vim 自动在 "-" 前面加了一个逗号。

要从中去掉一个字符用 "-="。例如要排除下划线：

```
:set iskeyword-=_
:set iskeyword
iskeyword=@,48-57,192-255,-
```

这回，逗号自动被删除了。

## 显 示 消 息 的 空 间

当 Vim 启动的时候，在屏幕底部有一行被用于显示消息。当消息很长的时候，多余的部分会被截断。这样你只能看到一部分。或者文字自动滚动，你要按 <Enter> 来继续。你可以给 'cmdheight' 选项赋一个值，用来设定显示消息所用的行数。例如：

```
:set cmdheight=3
```

这样意味着你用于编辑文字的空间少了，所以这实际上是一种折衷。

## 使用语法加亮

黑白的文字让人厌倦了，增加一些色彩能为你的文件带来生气。这不但看起来漂亮，
还能够提高你的工作效率。本章介绍如何使用不同颜色显示不同文本并把它打印出来。


下一章：|usr_07.txt|  编辑多个文件
前一章：|usr_05.txt|  选项设置
目录：|usr_toc.txt|

---

*06.1*  功能激活

一切从一个简单的命令开始：

```
:syntax enable
```

大多数情况下，这会让你的文件带上颜色。Vim 会自动检测文件的类型，并调用合适的
语法加亮。一下子注释变成蓝色，关键字变成褐色，而字符串变成红色了。这使你可以
很容易浏览整个文档。很快你就会发现，黑白的文本真的会降低你的效率！

如果你希望总能看到语法加亮，把 "syntax enable" 命令加入到 |vimrc| 文件中。

如果你想语法加亮只在支持色彩的终端中生效，你可以在 |vimrc| 文件中这样写：

```
if &t_Co > 1
  syntax enable
endif
```

如果你只想在 GUI 版本中有效，可以把 ":syntax enable" 放到你的 |gvimrc| 文件中。

---
-

*06.2*  颜色显示不出来或者显示出错误的颜色怎么办？

有很多因素会让你看不到颜色：

- 你的终端不支持彩色。
  这种情况下，Vim 会用粗体，斜体和下划线区分不同文字，但这不好看。你可能
  会希望找一个支持彩色的终端。对于 Unix，我推荐 XFree86 项目的 xterm：

|xfree-xterm|。

- 你的终端其实支持颜色，可是 Vim 不知道
  确保你的 $TERM 设置正确。例如，当你使用一个支持彩色的 xterm 终端：

  ```
  setenv TERM xterm-color
  ```

  或者（基于你用的控制台终端）

  ```
  TERM=xterm-color; export TREM
  ```

  终端名必须与你使用的终端一致。如果这还是不行，参考一下 |xterm-color|，
  那里介绍了一些使 Vim 显示彩色的方法（不仅是 xterm ）。

- 文件类型无法识别
  Vim 不可能识别所有文件，而且有时很难说一个文件是什么类型的。试一下这个
  命令：

  ```
  :set filetype
  ```

.
  如果结果是 "filetype="，那么问题就是出在文件类型上了。你可以手工指定
  文件类型：

  ```
  :set filetype=fortran
  ```

  要知道那些类型是有效的，查看一下 $VIMRUNTIME/syntax 目录。对于 GUI 版本，
  你还可以使用 Syntax 菜单。设置文件类型也可以通过 |modeline|，这样，
  在你每次编辑它的时候都执行语法加亮。例如，下面这一行可以用
  于 Makefile（把它放在接近文首和文末的地方）

  ```
  # vim: syntax=make
  ```

  你可能知道怎么检测自己的文件类型，通常是文件的扩展名（就是点后面的
  内容）参见 |new-filetype| 可以知道如何告诉 Vim 如何检查一种文件类型。

- 你的文件类型没有语法高亮定义
  你可以找一个相似的文件类型并人工设置为那种类型。如果你觉得不好，你可以
  自己写一个，参见 |mysyntaxfile|。

或者颜色是错的：

- 彩色的文字难以辨认
  Vim 自动猜测你使用的背景色。如果是黑的（或者其它深色的色彩），它会

用浅色作为前景色。如果是白的（或者其它浅色），它会使用深色作为前景色。如果 Vim 猜错了，文字就很难认了。要解决这个问题，设置一下 'background' 选项。对于深色：

```
:set background=dark
```

而对于浅色：

```
:set background=light
```

这两个命令必须在 ":syntax enable" 命令前调用，否则不起作用。如果要在这之后设置背景，可以再调用一下 ":syntax reset"。

- 在自下往上滚屏的过程中颜色显示不对。
Vim 在分析文本的时候不对整个文件进行处理，它只分析你要显示的部分。这样能省不少时间，但也会因此带来错误。一个简单的修正方法是敲 CTRL-L。或者往回滚动一下再回来。要彻底解决这个问题，请参见 |:syn-sync|。有些语法定义文件有办法自己找到前面的内容，这可以参见相应的语法定义文件。例如，|tex.vim| 中可以查到 Tex 语法定义。

---

*06.3*  使用不同颜色  *:syn-default-override*

如果你不喜欢默认的颜色方案，你可以选另一个配色方案。在 GUI 版本中可以使用 Edit/Color 菜单。你也可以使用这个命令：

```
:colorscheme evening
```

"evening" 是配色方案的名称。还有几种备选方案可以试一下。在 $VIMRUNTIME/colors 中可以找到这些方案。

等你确定了一种喜欢配色方案，可以把 ":colorscheme" 命令加到你的 |vimrc| 文件中。

你可以自己编写配色方案，下方如下法面：

1. 选择一种接近你的理想的配色方案。把这个文件拷贝到你自己的 Vim 目录中。在 Unix 下，可以这样：

```
!mkdir ~/.vim/colors
!cp $VIMRUNTIME/colors/morning.vim ~/.vim/colors/mine.vim
```

在 Vim 中完成的好处是可以利用 $VIMRUNTIME 变量。

2. 编辑这个配色方案，常用的有下面的这些条目：

| | |
|---|---|
| term | 黑白终端的属性 |
| cterm | 彩色终端的属性 |
| ctermfg | 彩色终端的前景色 |
| ctermbg | 彩色终端的背景色 |
| gui | GUI 版本属性 |
| guifg | GUI 版本的前景色 |
| guibg | GUI 版本的背景色 |

例如，要用绿色显示注释：

```
:highlight Comment ctermfg=green guifg=green
```

属性是 "bold"（粗体）和 "underline" （下划线）可以用于 "cterm" 和 "gui"。如果你两个都想用，可以用"bond,underline"。要获得详细信息， 请参考 |:highlight| 命令。

3. 告诉 Vim 总使用你这个配色方案。把如下语句加入你的 |vimrc| 中：

```
colorscheme mine
```

如果你要测试一下常用的配色组合，用如下命令：

```
:edit $VIMRUNTIME/syntax/colortest.vim
:source %
```

这样你会看到不同的颜色组合。你可以很容易的看到哪一种可读性好而且漂亮。

---

*06.4*  是否使用色彩

使用色彩显示文本会影响效率。如果你觉得显示得很慢，可以临时关掉这个功能：

```
:syntax clear
```

当你开始编辑另一个文件（或者同一个文件），色彩会重新生效。

*:syn-off*

如果你要完全关闭这个功能：

```
:syntax off
```

这个命令会停止对所有缓冲的所有语法加亮。

*:syn-manual*
如果你想只对特定的文件采使用语法加亮，可以使用这个命令：

:syntax manual

这个命令激活语法加亮功能，但不会在你开始编辑一个缓冲时自动生效（译者注：Vim 中，每个被打开的文件对应一个缓冲，后面的章节中你会接触到这方面的内容）。要在当前缓冲中使用加亮，需要设置'syntax'选项：

:set syntax=ON

---

*06.5*  带颜色打印 *syntax-printing*

在 MS-Windows 版本中，你可以用如下命令打印当前文件：

:hardcopy

这个命令会启动一个常见的打印对话框，你可以通过它选择打印机并作一些必要的设置。如果你使用的是彩色打印机，那么打印出来的色彩将与你在 Vim 中看到的一样。但如果你使用的是深色的背景，它的颜色会被适当调整，以便在白色地打印纸上看起来比较舒服。

下面几个选项可以改变 Vim 的打印行为：

'printdevice'
'printheader'
'printfont'
'printoptions'

要仅打印一定范围内的行，可以用可视模式选择需要打印的行在执行打印命令，例如：

v100j:hardcopy

"v" 启动可视模式，"100j" 向下选中 100 行，然后执行 ":hardcopy" 打印这些行。当然，你可以用其它命令选中这 100 行。

如果你有一台 PostScript 打印机，上面的方法也适合 Unix 系统。否则，你必须做一些额外的处理：你需要先把文件转换成 HTML 类型，然后用 Netscape 之类的浏览器打印。

如下命令把当前文件转换成 HTML 格式：

:source $VIMRUNTIME/syntax/2html.vim

你发现它会嘎吱嘎吱执行一阵子，（如果文件很大，这可能要花点时间）。之后，
Vim 会打开一个新的窗口并显示 HTML 代码。现在把这个文件存下来（存在哪都不要紧，

反正最后你要删掉它的)：

```
:write main.c.html
```

用你喜欢的浏览器打开这个文件，并通过它打印这个文件。如果一切顺利，这个输出应该与 Vim 中显示的一样。要了解更详细的信息，请参见 |2html.vim|。处理完后别忘了删掉那个 HTML 文件。
除了打印，你还可以把这个 HTML 文件，放到 WEB 服务器上，让其他人可以通过彩色文本阅读。

编辑多个文件

---

无论你有多少个文件，你都可以同时编辑它们而不需要退出 Vim。本章介绍如何定义一个文件列表，并基于这个列表工作，或者从一个文件跳转到另一个文件，又或者从一个文件中拷贝文字，并写入到另一个文件中。




下一章: |usr_08.txt|  分割窗口
前一章: |usr_06.txt|  使用语法高亮
目录: |usr_toc.txt|


---

## *07.1*  编辑另一个文件

在本章前，你都是为每一个文件启动一次 Vim 的。实际上还有其它办法。如下命令就可以在 Vim 中打开另一个文件:

```
:edit foo.txt
```

你可以用任何其它文件名取代上面的 "foo.txt"。Vim 会关闭当前文件并打开另一个。但如果当前文件被修改过而没有存盘， Vim 会显示错误信息而不会打开这个新文件:

```
E37: No write since last change (use ! to override)
```

(译者注: 在中文状态下显示:

E37: 已修改但尚未保存（可用 ！强制执行）

）

备注:
Vim 在每个错误信息的前面都放了一个错误号。如果你不明白错误信息的意思，可以从帮助系统中获得更详细的说明。对本例而言:

```
:help E37
```

出现上面的情况，你有多个解决方案。首先你可以通过如下命令保存当前文件:

```
:write
```

或者，你可以强制 Vim 放弃当前修改并编辑新的文件。这时应该使用强制修饰符 ！:

```
:edit! foo.txt
```

如果你想编辑另一个文件，但又不想马上保存当前文件，可以隐藏它:

```
:hide edit foo.txt
```

原来的文件还在那里，只不过你看不见。这将在 "|22.4|: 缓冲区列表" 中解释。

---

## *07.2* 文件列表

你可以在启动 Vim 的时候指定一堆文件。例如:

```
vim one.c two.c three.c
```

这个命令启动 Vim 并告诉它你要编辑三个文件。Vim 只显示第一个。等你编辑完第一个以后，用如下命令可以编辑第二个:

```
:next
```

如果你在当前文件中有未保存的修改，你会得到一个错误信息而无法编辑下一个文件。这个问题与前一节执行 ":edit" 命令的问题相同。要放弃当前修改:

```
:next!
```

但大多数情况下，你需要保存当前文件再进入下一个。这里有一个特殊的命令:

```
:wnext
```

这相当于执行了两个命令:

```
:write
:next
```

我 在 哪?

要知道当前文件在文件列表中的位置，可以注意一下文件的标题。那里应该显示类似 "(2 of 3)" 的字样。这表示你正在编辑三个文件中的第二个。

如果你要查看整个文件列表，使用如下命令:

```
:args
```

这是 "arguments" (参数) 的缩写。其输出应该象下面这样:

```
one.c [two.c] three.c
```

这里列出所有你启动 Vim 时指定的文件。你正在编辑的那一个，例如，"two.c"，会用中括号括起。

移 动 到 另 一 个 参 数

要回到前一个文件:

```
:previous
```

这个命令与 ":next" 相似，只不过它是向相反的方向移动。同样地，这个命令有一个快捷版本用于 "保存再移动":

```
:wprevious
```

要移动到列表中的最后一个文件:

```
:last
```

而要移动到列表中的第一个文件:

```
:first
```

不过，可没有 ":wlast" 或者 "wfirst" 这样的命令了。

你可以在 ":next" 和 ":previous" 前面加计数前缀。例如要向后跳两个文件:

```
:2next
```

自 动 保 存

当你在多个文件间跳来跳去进行修改，你要老记着用 ":write" 保存文件。否则你就会得到一个错误信息。 如果你能确定你每次都会将修改存盘的话，你可以让 Vim 自动保存文件:

```
:set autowrite
```

如果你编辑一个你不想自动保存的文件，你可以把功能关闭:

```
:set noautowrite
```

编 辑 另 一 个 文 件 列 表

你可以编辑另一个文件列表而不需要退出 Vim。用如下命令编辑另三个文件:

```
:args five.c six.c seven.h
```

或者使用通配符，就像在控制台上一样:

```
:args *.txt
```

Vim 会跳转到列表中的第一个文件。同样地，如果当前文件没有保存，你需要保存它，或者使用 ":args!" (加了一个 !) 放弃修改。

你 编 辑 了 最 后 一 个 文 件 吗?

*arglist-quit*

当你使用了文件列表，Vim 假定你想编辑全部文件，为了防止你提前退出，如果你还没有编辑过最后一个文件。当你退出的时候，Vim 会给如下错误信息:

```
E173: 46 more files to edit
```

如果你确实需要退出，再执行一次这个命令就行了 (但如果在两个命令间还执行了其它命令就无效了)。

---

## *07.3* 从一个文件跳到另一个文件

要在两个文件间快速跳转，按 CTRL-^ (美式英语键盘中 ^ 6 的上面)。例如:

```
:args one.c two.c three.c
```

现在你在 one.c。

```
:next
```

现在你在 two.c。现在使用 CTRL-^ 回到 one.c。再按一下 CTRL-^ 则回到 two.c。又按一下 CTRL-^ 你再回到 one.c。如果你现在执行:

```
:next
```

现在你在 three.c。注意 CTRL-^ 不会改变你在文件列表中的位置。只有 ":next" 和 ":previous" 才能做到这点。

你编辑的前一个文件称为 "轮换" 文件。如果你启动 Vim 而 CTRL-^ 不起作用，那可能是因为你没有轮换文件。

预 定 义 标 记

当你跳转到另一个文件后，有两个预定义的标记非常有用:

```
`"
```

这个标记使你跳转到你上次离开这个文件时的位置。
另一个标记记住你最后一次修改文件的位置:

```
`.
```

假设你在编辑 "one.txt"，在文件中间某个地方你用 "x" 删除一个字符，接着用 "G" 命令移到文件末尾，然后用 "w" 存盘。然后你又编辑了其它几个文件。你现在用 ":edit one.txt" 回到 "one.txt"。如果现在你用 `"，Vim 会跳转到文件的最后一行；而用 `. 则跳转到你删除字符的地方。即使你在文件中移动过，但在你修改或者离开文件前，这两个标记都不会改变。

文 件 标 记

在第四章，我们介绍过使用 "mx" 命令在文件中增加标记，那只在一个文件中有效。如果你编辑另一个文件并在那里加了标记，这些标记都是这个文件专用的。这样，每个文件都有一个自己的标记集，并只能在该文件中使用。

到此为止，我们都用小写字母的标记。实际上还可以使用大写字母标记，这种标记是全局的，它们可以在任何文件中使用。例如，你在编辑一个文件 "foo.txt"。在文件的中间 (50%) 并建立一个 J 标记 (J 表示甲):

```
50%mJ
```

现在编辑文件 "bar.txt" 并在文件的最后一行放一个标记 Y (Y 表示乙):

```
GmY
```

现在你可以使用 "`J" 命令跳回到 foo.txt 的中间。或者在另一个文件中输入 "`Y" 跳回到 bar.txt 的末尾。

文件标记会被一直记住直到被重新定义。这样，你可以在一个文件中留下一个标记，然后任意做一段时间的编辑，最后用这个标记跳回去。

让文件标记符和对应的位置建立一些关系常常是很有用的。例如，用 H 表示头文件 (Head File)，M 表示 Makefile 而 C 表示 C 的代码文件。

要知道一个标记在什么地方，在 ":marks" 命令中加上标记名作为参数即可:

```
:marks M
```

你还可以带多个参数:

```
:marks MCP
```

别忘了你还可以 CTRL-O 和 CTRL-I 在整个跳转序列中前后跳转。

---

## *07.4* 备份文件

通常 Vim 不会产生备份文件。如果你希望的话，执行如下命令就可以了:

```
:set backup
```

备份文件的文件名是在原始文件的后面加上一个 ~ 。如果你的文件名是 data.txt，则备份文件的文件名就是 data.txt~ 。

如果你不喜欢这个名字，你可以修改扩展名:

```
:set backupext=.bak
```

这会使用 data.txt.bak 而非 data.txt~。

还有一个相关选项是 'backupdir'。它指定备份文件的目录。默认情况是与原始文件的路径一致，这在很多情况下都是合适的。

> 备注:
> 如果 'backup' 选项没有置位而 'writebackup' 选项置了位，Vim 还是会创建备份文件的。但在文件编辑完后，这个备份文件会被自动删除。这个功能用于避免发生异常情况导致没有存盘 (磁盘满是最常见的情况；被雷击也是一种情况，不过很少发生)。

保 留 原 始 文 件

如果你在编辑源程序，你可能想在修改之前保留一个备份。但备份文件会在你存盘的时候被覆盖。这样它只能保留前一个版本，而不是最早的文件。

要让 Vim 保存一个原始的文件，可以设置 'patchmode' 选项。这个选项定义需要改动文件的第一个备份文件的扩展名。通常可以这样设:

```
:set patchmode=.orig
```

这样，当你第一次编辑 data.txt，作了修改并执行存盘，Vim 会保留一个名为 "data.txt.orig" 的原始文件。

如果你接着修改这个文件，Vim 会发现这个原始文件已经存在，并不再覆盖它。进一步的备份就存在 "data.txt~" (或者你设置的 'backupext' 指定的文件) 中。

如果你让 'patchmode' 设为空 (这是默认的情况)，则原始文件不会被保留。

---

## *07.5* 文件间拷贝文本

本节解释如何在文件间拷贝文本。我们从一个简单的例子开始。编辑一个你要拷贝文本的文件，把光标移到要拷贝的文本的开始处，用 "v" 命令启动可视模式，然后把光标移到要拷贝文本的结尾处，输入 "y" 拷贝文本。

例如，要拷贝上面这段文字，你可以执行:

```
:edit thisfile
/本节解释
vjjj$y
```

现在编辑你要粘贴文本的文件。把光标移到你要插入文本的地方。用 "p" 命令把文本粘贴到那里:

```
:edit otherfile
/There
p
```

当然，你可以用任何命令拷贝文本。例如，用 "V" 命令选中整行的内容。或者用 CTRL-V 选择一个矩形列块。或者使用 "Y" 拷贝一个单行，"yaw" 拷贝一个单词等。

"p" 命令把文本粘贴到光标之后，"P" 命令则粘贴到光标之前。注意， Vim 会记住你拷贝的是一整行还是一个列块，并用相同的方式把文本贴出来。

使 用 寄 存 器

当你需要拷贝一个文件的几个地方到另一个文件，用上面的方法，你就得反复在两个文件间跳来跳去。要避免这种情况，你可以把不同的文本拷贝到不同的寄存器中。

寄存器是 Vim 用来保存文本的地方。这里我们使用名称为 a 到 z 的寄存器（后面我们会发现还有其它寄存器）。让我们拷贝一个句子到 f 寄存器（f 表示 First）：

```
"fyas
```

"yas" 命令象以前说过的那样拷贝一个句子，而 "f 告诉 Vim 把文本拷贝到寄存器 f 。这必须放在拷贝命令的前面。

现在，拷贝三个整行到寄存器 l（l 表示 line）：

```
"l3Y
```

计数前缀也可以用在 "l 的前面。要拷贝一个文本列块到寄存器 b（代表 block）中：

```
CTRL-Vjjww"by
```

注意 "b 正好在 "y" 命令的前面，这是必须的。把它放在 "w" 命令的前面就不行。

现在你有了在寄存器 f，l 和 b 有三段文本。编辑另一个文件，并移到要插入文本的地方：

```
"fp
```

同样地，寄存器标识符 "f 必须在 "p" 命令的前面。

你可以用任何顺序粘贴寄存器的内容。并且，这些内容一直存在于寄存器中，直到你拷贝其它文件到这个寄存器中。这样，你可以粘贴任意多次。

删除文本的时候，你也可以指定寄存器。使用这个方法可以移动几处文本。例如，要删除一个单词并写到 w 寄存器中：

```
"wdaw
```

同样地，寄存器标识符必须在删除命令 "d" 的前面。

## 添 加 到 文 件

当你要在几个文件中收集文本，你可以用这个命令：

```
:write >> logfile
```

这个命令将文本写入到文件的末尾。这样实现了文件添加功能。这样使你免去了拷贝，编辑和拷贝的过程，省了两步。但你只能加到目标文件的末尾。

要只拷贝一部分内容，可以先用可视模式选中这些内容后在执行 ":write"。在第 10 章，你将学会选中一个行范围的办法。

## *07.6* 显示文件

有时，你只是想查看一个文件，而没打算修改它。有一个风险是你想都没想就输入了一个 "w" 命令。要避免这个问题，以只读模式编辑这个文件。

要用只读模式启动 Vim，可以使用这个命令:

```
vim -R file
```

在 Unix，如下命令可以完成相同的功能:

```
view file
```

现在，你就在用只读模式阅读这个文件 "file" 了。但你执行 ":w" 命令的时候，你会得到一个禁止写入的错误信息。

当你试图修改这个文件时，Vim 会给你一个告警提示:

```
W10: Warning: Changing a readonly file
```

即使这样，你的修改还是会被接纳的。有可能你只是想排列这些文本，以便阅读。

如果你确实要改动这个文件，在 write 命令前面加上 ! 可以强制写入。

如果你的确想禁止文件修改，用这个命令:

```
vim -M file
```

现在任何对文件的修改操作都会失败。例如，帮助文件就是这样的。如果你要在上面作修改，你会得到一个错误提示:

```
E21: Cannot make changes, 'modifiable' is off
```

你可以设置 -M 参数使 Vim 工作在只读模式。这个方式仍然取决于用户的意愿，因为你可以用下面的命令去掉这层保护:

```
:set modifiable
:set write
```

## *07.7* 修改文件名

编辑一个新文件的一个比较聪明的做法是使用一个现存的、其中大部分内容你都需要的文件。例如，你要写一个移动文件的程序，而你已经有一个用于拷贝的程序了，这样可以这样开始:

```
:edit copy.c
```

删除你不要的东西。现在你需要用一个新的文件名保存这个文件。":saveas" 命令就是为此设计的:

```
:saveas move.c
```

Vim 会用给定的名称保存文件，并开始编辑该文件。这样，下次你用 ":write"，写入的时候，被写入的就是 "move.c"。而 "copy.c" 不会被改变。

当你想改变当前文件的文件名，但不想立即保存它，用这个命令:

```
:file move.c
```

Vim 会把这个文件标记为 "未编辑"。这表示 Vim 知道你现在编辑的文件不是原来那个文件了。当你写这个文件的时候，你会得到如下错误信息:

```
E13: File exists (use ! to override)
```

这可以避免你不小心覆盖另一个文件。

分割窗口

显示两个不同的文件；或者同时显示一个文件的两个不同地方；又或者并排比较两个文件。这一切都可以通过分割窗口实现。




下一章: |usr_09.txt| 使用 GUI 版本
前一章: |usr_07.txt| 编辑多个文件
目录: |usr_toc.txt|

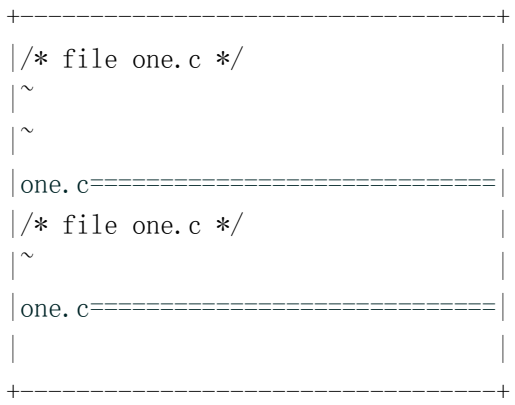

---

## *08.1* 分割窗口

打开新窗口最简单的命令如下:

```
:split
```

这个命令把屏幕分解成两个窗口并把光标置于上面的窗口中:




你可以看到显示同一个文件的两个窗口。带 "====" 的行是状态条，用来显示它上面的窗口的信息。(在实际的屏幕上，状态条用反色显示)

这两个窗口允许你同时显示一个文件的两个部分。例如，你可以让上面的窗口显示变量定义而下面的窗口显示使用这些变量的代码。

CTRL-W w 命令可以用于在窗口间跳转。如果你在上面的窗口，它会跳转到下面的窗口，如果你在下面的窗口，它会跳转到上面的窗口。(CTRL-W CTRL-W 可以完成相同的功能这是为了避免你有时按第二次的时候从 CTRL 键上缩手晚了。)

关 闭 窗 口

以下命令用于关闭窗口:

```
:close
```

实际上，任何退出编辑的命令都可以关闭窗口，象 ":quit" 和 "ZZ" 等。但 "close" 可以避免你在剩下一个窗口的时候不小心退出 Vim 了。

关 闭 所 有 其 它 窗 口

如果你已经打开了一整套窗口，但现在只想编辑其中一个，如下命令可以完成这个功能:

```
:only
```

这个命令关闭除当前窗口外的所有窗口。如果要关闭的窗口中有一个没有存盘，Vim 会显示一个错误信息，并且那个窗口不会被关闭。

---

## *08.2* 用另一个文件分割窗口

下面命令打开另一个窗口并用该窗口编辑另一个指定的文件:

```
:split two.c
```

如果你在编辑 one.c，则命令执行的结果是:

```
+--------------------------------+
|/* file two.c */                |
|~                               |
|~                               |
|two.c============================|
|/* file one.c */                |
|~                               |
|one.c============================|
|                                |
+--------------------------------+
```

要打开窗口编辑一个新文件，可以使用如下命令:

```
:new
```

你可以重复使用 ":split" 和 ":new" 命令建立任意多的窗口。

---

## *08.3* 窗口大小

:split 命令可以接受计数前缀。如果指定了这个前缀，这个数将作为窗口的高度。例如如下命令可以打开一个三行的窗口并编辑文件 alpha.c:

```
:3split alpha.c
```

对于已经打开的窗口，你可以用有几种方法改变它的大小。如果你有鼠标，很简单: 把鼠标指针移到分割两个窗口的状态栏上，上下拖动即可。

要扩大窗口:

```
CTRL-W +
```

要缩小窗口:

```
CTRL-W -
```

这两个命令接受计数前缀用于指定扩大和缩小的行数。所以 "4 CTRL-W +" 会使窗口增高 4 行。

要把一个窗口设置为指定的高度，可以用这个命令：

```
{height}CTRL-W _
```

就是先输入一个数值，然后输入 CTRL-W 和一个下划线（在美式英语键盘中就是 Shift 加上 "-"）。

要把一个窗口扩展到尽可能大，可以使用无计数前缀的 CTRL-W _ 命令。

使 用 鼠 标

在 Vim 中，你可以用键盘很快完成很多工作。但很不幸，改变窗口大小要敲不少键。在这种情况下，使用鼠标会更快一些。把鼠标指针移到状态条上，按住左键并拖动。状态条会随之移动，这会使一个窗口更大一个更小。

选 项

'winheight' 选项设置最小的期望窗口高度而 'winminheight' 选项设置最小的 "硬性" 高度。

同样，'winwidth' 设置最小期望宽度而 'winminwidth' 设置最小硬性宽度。

'equalalways' 选项使所有的窗口在关闭或者打开新窗口的时候总保持相同大小。

---

## *08.4* 垂直分割

":split" 命令在当前窗口的上面建立窗口。要在窗口左边打开新窗口，用这个命令：

```
:vsplit
```

或者

```
:vsplit two.c
```

这个命令的结果如下：

```
+-----------------------------------+
|/* file two.c */   |/* file one.c */  |
|~                  |~                 |
|~                  |~                 |
|~                  |~                 |
|two.c==============one.c=============|
|                                      |
+-----------------------------------+
```

实际中，中间的竖线会以反色显示。这称为垂直分割线。它左右分割一个窗口。

还有一个 "vnew" 命令，用于打开一个垂直分割的新窗口。还有一种方法是:

```
:vertical new
```

"vertical" 命令可以放在任何分割窗口的命令的前面。这会在分割窗口的时候用垂直分割取代水平分割。(如果命令不分割窗口，这个前缀不起作用)。

在 窗 口 间 跳 转

由于你可以用垂直分割和水平分割命令打开任意多的窗口，你就几乎能够任意设置窗口的布局。接着，你可以用下面的命令在窗口之间跳转:

```
CTRL-W h        跳转到左边的窗口
CTRL-W j        跳转到下面的窗口
CTRL-W k        跳转到上面的窗口
CTRL-W l        跳转到右边的窗口

CTRL-W t        跳转到最顶上的窗口
CTRL-W b        跳转到最底下的窗口
```

你可能已经注意到这里使用移动光标一样的命令用于跳转窗口。如果你喜欢，改用方向键也行。

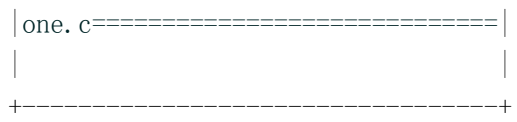
还有其它命令可以跳转到别的窗口，参见: |Q_wi|。

---

## *08.5* 移动窗口

你已经分割了一些窗口，但现在的位置不正确。这时，你需要一个命令用于移动窗口。例如，你已经打开了三个窗口，象这样:

```
+---------------------------------+
|/* file two.c */                 |
|~                                |
|~                                |
|two.c============================|
|/* file three.c */               |
|~                                |
|~                                |
|three.c==========================|
|/* file one.c */                 |
|~                                |
```

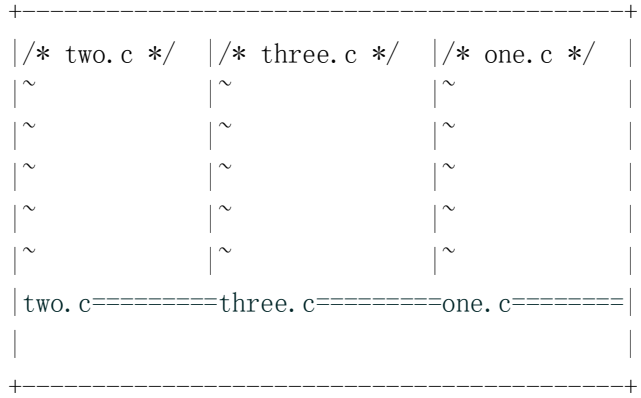



显然，最后一个窗口应该在最上面。移动到那个窗口（用 CTRL-W w）并输入如下命令：

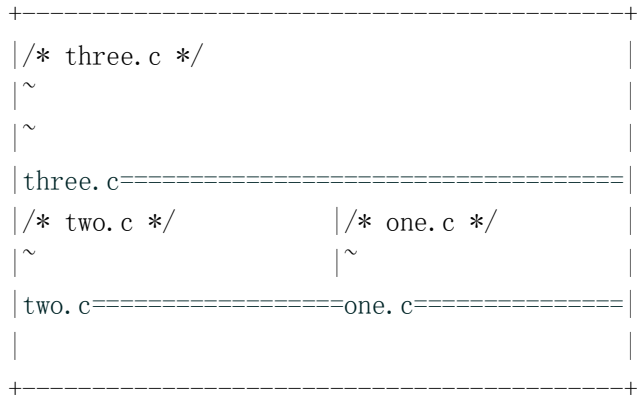
```
CTRL-W K
```

这里使用大写的 K。这样窗口将被移到最上面。你可以注意到，这里又用 K 表示向上移动了。

如果你用的是垂直分割，CTRL-W K 会使当前窗口移动到上面并扩展到整屏的宽度。假设你的布局如下：




当你在中间的窗口（three.c）中使用 CTRL-W K 后，结果会是：




还有三个相似的命令（估计你已经猜出来了）：

```
CTRL-W H        把当前窗口移到最左边
CTRL-W J        把当前窗口移到最下边
CTRL-W L        把当前窗口移到最右边
```

## *08.6*  对所有窗口执行命令

你打开了几个窗口，现在你想退出 Vim，你可以分别关闭每一个窗口。更快的方法是:

```
:qall
```

这表示 "quit all"（全部退出）。如果任何一个窗口没有存盘，Vim 都不会退出。同时光标会自动跳到那个窗口，你可以用 ":write" 命令保存该文件或者 ":quit!" 放弃修改。

如果你知道有窗口被改了，而你想全部保存，则执行如下命令:

```
:wall
```

这表示 "write all"（全部保存）。但实际上，它只会保存修改过的文件。Vim 知道保存一个没有修改过的文件是没有意义的。

另外，还有 ":qall" 和 "wall" 的组合命令:

```
:wqall
```

这会保存所有修改过的文件并退出 Vim 。

最后，下面的命令由于退出 Vim 并放弃所有修改:

```
:qall!
```

注意，这个命令是不能撤消的。

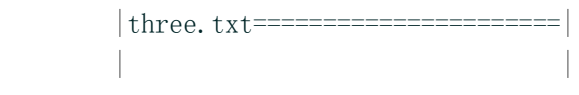
### 为所有的参数打开窗口

要让 Vim 为每个文件打开一个窗口，可以使用 "-o" 参数:

```
vim -o one.txt two.txt three.txt
```

这个结果会是:

```
+-----------------------------+
|file one.txt                 |
|~                            |
|one.txt======================|
|file two.txt                 |
|~                            |
|two.txt======================|
|file three.txt               |
|~                            |
```




"-o" 参数用于垂直分割窗口。

如果 Vim 已经启动了，可以使用 ":all" 命令为参数列表中的每个文件打开一个窗口。":vertical all" 以垂直分割的方法打开窗口。

---

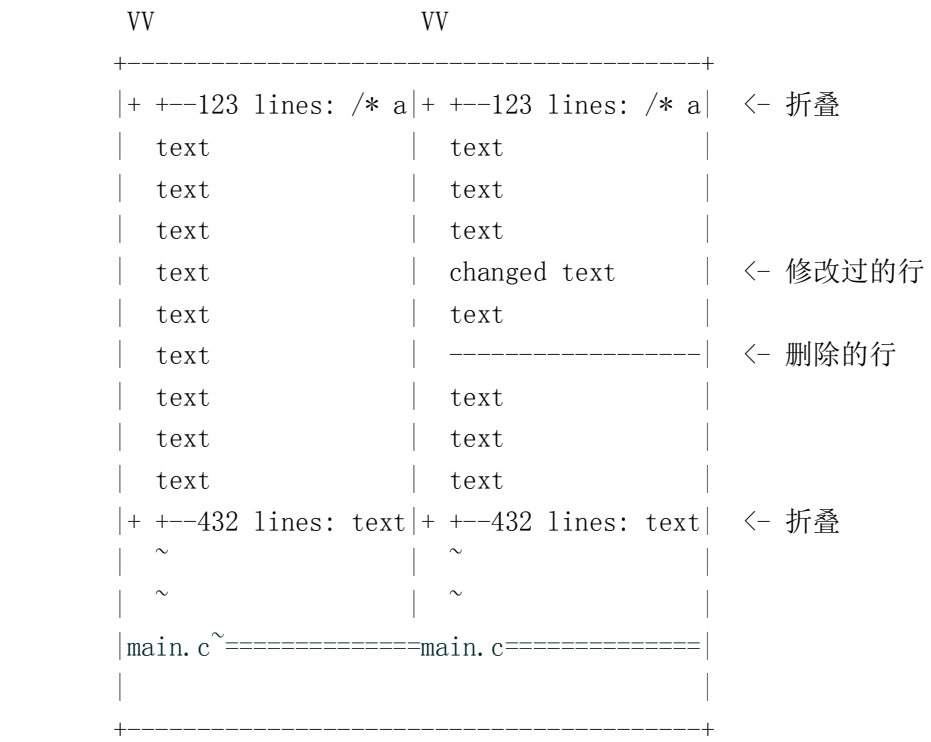
## *08.7* 用 vimdiff 显示文件差异

有一种特殊的启动 Vim 的方法可以用来显示两个文件的差异。让我们打开一个 "main.c" 并插入一些字符。在设置了 'backup' 选项的情况下保存这个文件，以便产生 "main.c~" 备份文件。

在命令行中输入如下命令：(不是在 Vim 中)

```
vimdiff main.c~ main.c
```

Vim 会用垂直分割的方式打开两个文件。你只能看到你修改过的地方和上下几行的地方。




(这幅图没有显示出高亮效果，可以使用 vimdiff 命令看到更好的效果)

那些没有修改的行会被折叠成一行，这称为 "关闭的折叠" (closed fold)。上图中由 "<- 折叠" 标记的行就是一个用一行表示 123 行的折叠。这些行在两个文件中完全相同。

标记为 "<- 修改过的行" 被高亮显示，而增加的行被用另一种颜色表示。这可以很清楚地表示出两个文件间的不同。

被删除的行在 main.c 窗口中用 "---" 显示，如图中用 "<- 删除的行" 标记的行。这些字符并不是真的存在。它们只是用于填充 main.c，以便与另一个窗口对齐。

## 折 叠 栏

每个窗口在左边都有一个颜色略有不同的显示栏，图中标识为 "VV"。你会发现每个折叠在那个位置都有一个加号。把鼠标移到那里并按左键可以打开那个折起，从而让你看到里面的内容。

对于打开的折叠，折叠栏上会出现一个减号。如果你单击那个减号，折叠会被重新关闭。

当然，这只能在你有鼠标的情况下使用。如果你没有，可以用 "zo" 打开一个折叠。关闭使用 "zc"。

## 用 Vim 做 比 较

启动比较模式的另一种方法从 Vim 内部开始：编辑 "main.c" 文件，然后分割窗口显示区别：

```
:edit main.c
:vertical diffsplit main.c~
```

":vertical" 命令使窗口用垂直的方式分割。如果你不写这个命令，结果会变成水平分割。

如果你有一个当前文件的补丁或者 diff 文件，你可以用第三种方法启动比较模式：先编辑这个文件，然后告诉 Vim 补丁文件的名称：

```
:edit main.c
:vertical diffpatch main.c diff
```

警告：补丁文件中必须仅包括为一个目标文件所做的补丁，否则你可能会得到一大堆错误信息。还可能有些你没打算打补丁的文件也被打了补丁。

补丁功能只改变内存中的文件备份，不会修改你硬盘上的文件（除非你决定写入改动）。

## 滚 动 绑 定

当文件中有很多改动时，你可以用通常的方式滚动屏幕。Vim 会尽可能保持两个文件对齐，以便你可以并排看到文件的区别。

如果暂时想关闭这个特性，使用如下命令：

```
:set noscrollbind
```

## 跳 转 到 修 改 的 地 方

如果你通过某种方法取消了折叠功能，可能很难找到有改动的地方。使用如下命令可以跳转到下一个修改点：

```
]c
```

反向跳转为：

```
[c
```

加上一个计数前缀可以跳得更远。

## 消 除 差 异

你可以把文本从一个窗口移到另一个，并以此来消除差异，或者为其中一个文件中增加几行。Vim 有时可能无法及时更新高亮显示。要修正这种问题，使用如下命令：

```
:diffupdate
```

要消除差异，你可以把一个高亮显示的块从一个窗口移动到另一个窗口。以上面的 "main.c" 和 "main.c~" 为例，把光标移到左边的窗口，在另一个窗口中被删除的行的位置，执行如下命令：

```
:dp
```

这将把文字从左边拷到右边，从而消除两边的差异。"dp" 代表 "diff put"。

你也可以反过来做：把光标移到右边的窗口，移到被 "改动" 了的行上，然后执行如下命令：

```
:do
```

这把文本从左边拷到右边，从而消除差异。
由于两个文件已经没有区别了，Vim 会把所有文字全部折叠起来。"do" 代表 "diff obtain"。本来用 "dg" (diff get) 会更好。可是它已经有另外的意思了 ("dgg" 删除从光标为止到首行的所有文本)。

要了解更多的比较模式的内容，参见 |vimdiff|。

## *08.8* 杂项

'laststatus' 选项用于指定什么时候对最后一个窗口显示状态条:

| | |
|---|---|
| 0 | 永远不 |
| 1 | 只有用分割窗口的时候（默认） |
| 2 | 永远有 |

很多编辑另一个文件的命令都有一个使用分割窗口的变体。对于命令行命令，这通过前置一个 "s" 实现。例如 ":tag" 用来跳到一个标记，"stag" 就会分割出一个新窗口并跳到那个标记。

对于普通模式，前置一个 CTRL-W 可以完成这个功能。例如，CTRL-^ 跳到轮换文件，而 CTRL-W CTRL-^ 打开一个新窗口并编辑轮换文件。

'splitbelow' 选项可以让新的窗口出现在当前窗口的下面。'splitright' 选项让垂直分割的窗口出现在当前窗口的右边。

打开一个新窗口时可以在命令前加上一个修饰符说明新窗口应该出现在什么地方:

| | |
|---|---|
| :leftabove {cmd} | 当前窗口的左上方 |
| :aboveleft {cmd} | 同上 |
| :rightbelow {cmd} | 当前窗口的右下方 |
| :belowright {cmd} | 同上 |
| :topleft {cmd} | 整个 Vim 窗口的最上面或者最左边 |
| :botright {cmd} | 整个 Vim 窗口的最下面或者最右边 |

## *08.9* 标签页

你会注意到窗口永远不会重叠。这意味着屏幕空间很快会用完。这个问题的解决方法叫做标签页。

假设你正在编辑文件 "thisfile"。下面的命令可以建立新的标签页:

```
:tabedit thatfile
```

这会在一个窗口中编辑文件 "thatfile"，这个窗口会占满整个 Vim 窗口。你会注意到在顶部有一个含有两个文件名的横条:

```
+---------------------------------+
| thisfile | /thatfile/ _________X|    (thatfile 用加粗字体出现)
|/* thatfile */                   |
```

```
|that                              |
|that                              |
|~                                 |
|~                                 |
|~                                 |
|                                  |
+----------------------------------+
```

现在，你拥有了两个标签页。第一个是文件 "thisfile" 的窗口，第二个是文件 "thatfile" 的窗口。这就像是两张重叠的纸，它们所带的的标签露在外面，显示其文件名。

现在，使用鼠标单击顶端的 "thisfile"。结果是

```
+----------------------------------+
| /thisfile/ | thatfile _________X|   (thisfile 用加粗字体出现)
|/* thisfile */                    |
|this                              |
|this                              |
|~                                 |
|~                                 |
|~                                 |
|                                  |
+----------------------------------+
```

你可以通过单击顶端的标签切换标签页。如果没有鼠标或者不想用它，可以使用 "gt" 命令。助记符: Goto Tab。

现在，让我们通过下面的命令建立另一个标签页:

```
:tab split
```

这会建立一个新的标签页，包含一个窗口，编辑和刚才所在窗口中的缓冲区相同的缓冲区:

```
+-----------------------------------+
| thisfile | /thisfile/ | thatfile __X|  (thisfile 用加粗字体出现)
|/* thisfile */                      |
|this                               |
|this                               |
|~                                  |
|~                                  |
|~                                  |
|                                   |
+-----------------------------------+
```

在任何打开窗口的 Ex 命令前面，你都可以放上 ":tab"。这个窗口在新标签页中打开。另一个例子:

```
:tab help gt
```

它将在新的标签页中显示关于 "gt" 的帮助。

使用标签页可以完成更多的工作:

- 在末尾标签后面的空白处单击鼠标
  选择下个标签页，同 "gt"。

- 在右上角的 "X" 处单击鼠标
  关闭当前标签页，除非当前标签页中的改变没有保存。

- 在标签行上双击鼠标
  建立新标签页。

- "tabonly" 命令
  关闭除了当前标签页以外的所有标签页，除非其它标签页中的改变没有保存。

关于标签页更多的信息，参见 |tab-page|。

使用 GUI 版本

Vim 能在一般的终端中很好地工作。GVim 则可以完成相同，甚至更多的功能。Gvim 能提供菜单，工具条，滚动条和其它东西。本章介绍这些额外的功能。




下一章: |usr_10.txt|  做大修改
前一章: |usr_08.txt|  分割窗口
目录: |usr_toc.txt|


---

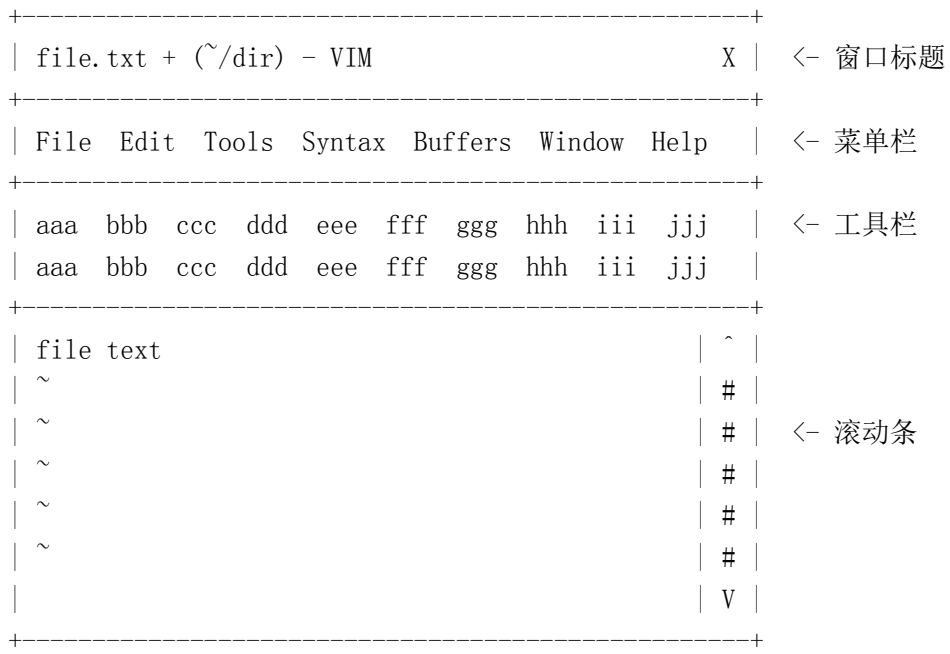
## *09.1*  GUI 版本的组件

你可以在你的桌面上放一个启动 gVim 的图标。此外，下面的任一个命令也可以启动 gVim:

```
gvim file.txt
vim -g file.txt
```

如果这样不行，可能是因为你的 Vim 不支持 GUI 版本特性。你需要先安装一个合适的版本。

执行命令后，Vim 会打开一个窗口，并显示文件 "file.txt"。窗口的样子取决于 Vim 的版本。一般是下面这个样子 (尽可能地用 ASCII 码展示):




最大的一片空间是文件的内容。这部分与终端上看到的是一样的，只是颜色和字体可能有一点差别。

## 窗 口 标 题

窗口最顶上是窗口标题。这由你的窗口系统绘制。Vim 会在这个标题上显示当前文件的相关信息。首先显示的是文件名，然后是一个特殊字符，最后是用括号括住的目录名。下面的是这些特殊字符的含义:

| | |
|---|---|
| - | 文件不能被修改 (例如帮助文件) |
| + | 已经被修改过 |
| = | 文件只读 |
| =+ | 文件只读，但仍被修改过 |

如果没有显示任何特殊字符，表示这是一个普通的，没有改过的文件。

## 菜 单 栏

你知道菜单是怎么工作的，是吧？ Vim 有些通用的菜单，外加一些特别的。逐个看看，猜测一下这些菜单都可以用来干什么。另一个相关的子菜单是 Edit/Global，你可以在那里找到这些菜单项：

```
Toggle Toolbar          使工具条可见/不可见
Toggle Bottom Scrollbar 使底部的滚动条可见/不可见
Toggle Left Scrollbar   使左边的滚动条可见/不可见
Toggle Right Scrollbar  使右边的滚动条可见/不可见
```

在大多数系统里，你可以把菜单 "撕下来"。选中菜单最上面的菜单栏，就是那个看起来象条虚线的。这样你可以得到一个分离的菜单，里面包括了所有菜单项。它会一直挂在那里，直到你关闭它。

## 工 具 栏

这里包括使用最频繁的操作的图标。希望这些图标功能显而易见。另外，每个图标都支持 "工具提示"（把鼠标移上去停一会儿就能看见这个提示）

"Edit/Global Settings/Toggle Toolbar" 菜单项可以关闭工具条。如果你从来都不使用工具条，可以在 vimrc 文件中加上：

```
:set guioptions-=T
```

这个命令从 'guioptions' 中删除 "T" 标记。其它 GUI 部件也可以通过这种方法激活或关闭。参见这个选项的相关帮助。

## 滚 动 条

默认情况下，右边会有一个滚动条，它的作用是很明显的。当你分割窗口的时候，每个窗口都会有自己的滚动条。

你可以通过 "Edit/Global Settings/Toggle Bottom Scrollbar" 来启动一个水平滚动条。这在比较模式或没有设置 'wrap' 时非常有用（后面有更多描述）。

在使用垂直分割的时候，只有右边的窗口有滚动条，但当你把光标移到左边的窗口上，右边的滚动条会对这个窗口起作用，这需要一些时间去适应。

当你使用垂直分割的时候，可以考虑把滚动条放在左边。这可以通过菜单激活，或者使用 'guioptions' 选项：

```
:set guioptions+=l
```

这是在 'guioptions' 中增加 'l' 标志位。

## *09.2* 使用鼠标

标准是好东西。在微软的 Windows 操作系统中，你可以用标准模式选中文本。X Windows 也有一套使用鼠标的标准。非常不幸，这两套标准是不同的。

幸运的是，你可以定制 Vim。你可以让你的鼠标行为象 X Windows 或者象微软 Windows 的鼠标。下面的命令使鼠标用起来象 X Windows:

```
:behave xterm
```

而如下命令使鼠标用起来象微软 Windows:

```
:behave mswin
```

在 UNIX 操作系统中，默认的鼠标行为是 xterm。而默认的微软 Windows 系统的鼠标行为是在安装的时候选定的。要了解这两种行为的详细信息，请参考 |:behave|。下面是一些摘要:

XTERM 鼠 标 行 为

| | |
|---|---|
| 左键单击 | 定位光标 |
| 左键拖动 | 在可视模式下选中文本 |
| 中键单击 | 从剪贴板中粘贴文本 |
| 右键单击 | 把选中的文本扩展到当前的光标位置 |

微软 Windows 鼠标行为

| | |
|---|---|
| 左键单击 | 定位光标 |
| 左键拖动 | 在选择模式下选中文本（参见 |09.4|） |
| 按住 Shift，左键单击 | 把选中的文本扩展到当前的光标位置 |
| 中键单击 | 从剪贴板中粘贴文本 |
| 右键单击 | 显示一个弹出式菜单 |

可以进一步定制鼠标。请参见下面的选项:

| | |
|---|---|
| 'mouse' | 鼠标的使用模式 |
| 'mousemodel' | 鼠标单击的效果 |
| 'mousetime' | 双击的间隔允许时间 |
| 'mousehide' | 输入的时候隐藏鼠标 |
| 'selectmode' | 鼠标启动可视模式还是选择模式 |

## *09.3*  剪贴板

|04.7| 节已经介绍过剪贴板的基本使用了。这里有一个重要的地方要解释一下：对于 X-windows 系统，有两个地方可以在程序间交换文本，而 MS-Windows 不是这样的。

在 X-Windows，有一个 "当前选择区" 的概念。它表示正被选中的文本。在 Vim 中，这表示可视区（假定你正使用默认的设置）。不需要任何其它操作，你就可以把这些文本贴到别的程序中。

例如，你用鼠标在本文中选中一些文本。Vim 会自动切换到可视模式，并高亮这些文本。现在启动另一个 gVim，（由于没有指定文件名，它会显示出一个空窗口）。点击鼠标中键。被选中的文本就会被贴进来。

"当前选择区" 会一直保持有效直到你选中其它文本。在另一个窗口中粘贴文本后，在这个窗口中选中一些文字，你会发现上一个窗口中选中的文字显示的方法跟原来有些区别了，这表示这些文字已经不是 "当前选择区" 了。

你不一定要用鼠标来选中文字，用键盘的 "可视" 命令也能达到相同的效果。

### "真" 剪 贴 板

对于另一个交换文本的地方，我们称之为 "真" 剪贴板以避免与上面的 "当前选择区" 混淆。通常 "当前选择区" 和 "真" 剪贴板都称为剪贴板，你需要习惯这些名称。

要把文字拷贝到真剪贴板，在一个 gVim 中选中一些文本，然后执行菜单命令 Edit/Copy。这样文字就被拷贝到真剪贴板了。剪贴板的内容是不可见的，除非你使用特别的显示程序，例如 KDE 的 klipper 程序。

现在，切换到另一个 gVim，把光标停在某个位置，然后执行菜单命令 Edit/Paste 菜单。你会看到真剪贴板中的内容被插入到当前的光标位置。

### 使 用 两 种 剪 贴 板

这种同时使用 "当前选择区" 和 "真剪贴板" 的操作方式听起来很乱。但这是很有用的。我们通过一个例子来说明。用 gVim 打开一个文件并执行如下命令：

- 在可视模式下选中两个词
- 使用 Edit/Copy 菜单把这些词拷到剪贴板
- 再用可视模式选中另一个词
- 执行 Edit/Paste 菜单命令。这样第二次选中的词会被前面剪贴板中的词代替。
- 把鼠标移到另一个地方按中键，你会发现你刚被覆盖的单词被粘贴到新的位置。

如果你小心使用 "当前选择区" 和 "真剪贴板" 两个工具，你可以完成很多很有用的工作。

### 使 用 键 盘

如果你不喜欢使用鼠标，你可以通过两个寄存器来使用 "当前选择区" 和 "真剪贴板" 两个剪贴板。"* 寄存器用于表示当前选择区。

要使文本变成 "当前选择区"，只要使用可视模式即可。例如，要选中一整行只要输入 "V"。

要拷贝当前选择区的内容:

```
"*P
```

注意这里 "P" 是大写，表示把文字拷贝到光标的前面。

"+ 寄存器用于真剪贴板。例如，要把当前光标位置到行末的文本拷到真剪贴板:

```
"+y$
```

记得吧，"y" 是 yank，这是 Vim 的拷贝命令。

要把真剪贴板的内容拷到光标前面:

```
"+P
```

这与 "当前选择区" 一样，只是用 (+) 寄存器取代了 (*) 寄存器。

---

## *09.4* 选择模式

现在介绍一些在 MS-Windows 中比在 X-Windows 中更常被使用的东西 (但在两个系统上都可用)。你已经了解可视模式了。选择模式与可视模式相似，也是用来选中文字的。但有一个显著区别: 当输入文本的时候，在选择模式下，被选中的文字将被替换成新输入的文字。

要启用选择模式，先要激活它 (对于 MS-Windows，可能已经激活了，不过多做一次也没什么):

```
:set selectmode+=mouse
```

现在用鼠标选中一些文本，这些文本会好像可视模式一样被高亮。现在敲入一个字母。被选中的文本被删除，替换成新的字母。现在已经是插入模式了，你可以继续输入。

由于输入普通文本导致选中的文字被删除，这时你不能使用 "hjkl"，"w" 等移动命令。这时可以使用 "Shift" 加功能键。<S-Left> (shift 键加左箭头) 使光标左移。选中的文字象可视模式一样被扩展或者减少。其它箭头起的作用你也可以猜到了，<S-End> 和 <S-Home> 也一样。

你可以通过 'selectmode' 选项修改选择模式的工作方式。

做大修改

---

第四章我们已经介绍过作小修改的方法了。本章开始介绍如何重复多次修改和如何改动大量的文字。这将包括使用可视模式处理一些文本块，还有使用一个外部程序去完成非常复杂的功能。




下一章: |usr_11.txt| 从崩溃中恢复
前一章: |usr_09.txt| 使用 GUI 版本
目录: |usr_toc.txt|


---

## *10.1* 记录与回放命令

"." 命令重复前一个修改操作。但如果你需要作一些更复杂的操作它就不行了。这时，记录命令就变得很有效。这需要三个步骤:

1. "q{register}" 命令启动一次击键记录，结果保存到 {register} 指定的寄存器中。寄存器名可以用 a 到 z 中任一个字母表示。
2. 输入你的命令。
3. 键入 q（后面不用跟任何字符）命令结束记录。

现在，你可以用 "@{register}" 命令执行这个宏。

现在看看你可以怎么用这些命令。假设你有如下文件名列表:

```
stdio.h
fcntl.h
unistd.h
stdlib.h
```

而你想把它变成这样:

```
#include "stdio.h"
```

```
#include "fcntl.h"
#include "unistd.h"
#include "stdlib.h"
```

先移动到第一行，接着执行如下命令:

```
qa                      启动记录，并使用寄存器 a
^                       移到行首
i#include "<Esc>        在行首输入 #include "
$                       移到行末
a"<Esc>                 在行末加上双引号 (")
j                       移到下一行
q                       结束记录
```

现在，你已经完成一次复杂的修改了。你可以通过重复三次 "@a" 完成余下的修改。

"@a" 命令可以通过计数前缀修饰，使操作重复指定的次数。在本例中，你可以输入:

```
3@a
```

## 移 动 并 执 行

你可能有多个地方需要修改。只要把光标移动到相应的位置并输入 "@a" 命令即可。如果你已经执行过一次，你可以用 "@@" 完成这个操作，这更容易输入一些。例如，你上次使用 "@b" 命令引用了寄存器 b，下一个 "@@" 命令将使用寄存器 b。

如果你对回放命令和 "." 命令作一个比较，你会发现几个区别。首先，"." 只能重复一次改动。而在上例中，"@a" 可以重复多次改动，还能够执行移动操作。第二， "." 只能记住最后一次变更操作。而寄存器执行命令允许你记录任何操作并使用象 "@a" 这样的命令回放这些被记录的操作。最后，你可以使用 26 个寄存器，因此，你可以记录多达 26 个命令序列。

## 使 用 寄 存 器

用来记录操作的寄存器与你用来拷贝文本的寄存器是相同的。这允许你混合记录操作和其它命令来操作这些寄存器。

假设你在寄存器 n 中记录了一些命令。当你通过 "@n" 执行这些命令时，你发现这些命令有些问题。这时你可以重新录一次，但这样你可能还会犯其它错误。其实，你可以使用如下窍门:

```
G                       移到行尾
o<Esc>                  建立一个空行
"np                     拷贝 n 寄存器中的文本，你的命令将被拷到整个文
```

| | |
|---|---|
| | 件的结尾 |
| {edits} | 象修改普通文本一样修改这些命令 |
| 0 | 回到行首 |
| "ny$ | 把正确的命令拷贝回 n 寄存器 |
| dd | 删除临时行 |

现在你可以通过 "@n" 命令执行正确的命令序列了。(如果你记录的命令包括换行符，请调整上面例子中最后两行的操作来包括所有的行。)

追 加 寄 存 器

到此为止，我们一直使用小写的寄存器名。要附加命令到一个寄存器中，可以使用大写的寄存器名。

假设你在寄存器 c 中已经记录了一个修改一个单词的命令。它可以正常工作，但现在你需要附加一个搜索命令以便找到下一个单词来修改。这可以通过如下命令来完成:

```
qC/word<Enter>q
```

启动 "qC" 命令可以对 c 寄存器追加记录。由此可见，记录到一个大写寄存器表示附加命令到对应的小写寄存器。

这种方法在宏记录，拷贝和删除命令中都有效。例如，你需要把选择一些行到一个寄存器中，可以先这样拷贝第一行:

```
"aY
```

然后移到下一个要拷贝的地方，执行:

```
"AY
```

如此类推。这样在寄存器 a 中就会包括所有你要拷贝的所有行。

---

## *10.2* 替换 *find-replace*

":substitute" 命令使你可以在连续的行中执行字符串替换。下面是这个命令的一般形式:

```
:[range]substitute/from/to/[flags]
```

这个命令把 [range] 指定范围中的字符串 "from" 修改为字符串 "to"。例如，你可以把连续几行中的 "Professor" 改为 "Teacher"，方法是:

```
:%substitute/Professor/Teacher/
```

备注:
很少人会把整个 ":substitute" 命令完整敲下来。通常，使用命令的缩写形式 ":s" 就行了。下文我们将使用这个缩写形式。

命令前面的 "%" 表示命令作用于全部行。如果不指定行范围，":s" 命令只作用在当前行上。|10.3| 将对 "行范围" 作深入的介绍。

默认情况下，":substitute" 命令只对某一行中的第一个匹配点起作用。例如，前面例子中会把行:

```
Professor Smith criticized Professor Johnson today.
```

修改成:

```
Teacher Smith criticized Professor Johnson today.
```

要对行中所有匹配点起作用，你需要加一个 g (global，全局) 标记。下面命令:

```
:%s/Professor/Teacher/g
```

对上面例子中的句子的作用效果如下:

```
Teacher Smith criticized Teacher Johnson today.
```

":s" 命令还支持其它一些标志位，包括 "p" (print，打印)，用于在命令执行的时候打印出最后一个被修改的行。还有 "c" (confirm，确认) 标记会在每次替换前向你询问是否需要替换。执行如下命令:

```
:%s/Professor/Teacher/c
```

Vim 找到第一个匹配点的时候会向你提示如下:

```
replace with Teacher (y/n/a/q/l/^E/^Y)?
```

(中文翻译如下:

```
替换为 Teacher 么 (y/n/a/q/l/^E/^Y)?
```

)

这种时候，你可以输入如下回答中的一个:

| | |
|---|---|
| y | Yes，是；执行替换 |
| n | No，否；跳过 |

| | |
|---|---|
| a | All，全部；对剩下的匹配点全部执行替换，不需要再确认 |
| q | Quit，退出；不再执行任何替换 |
| l | Last，最后；替换完当前匹配点后退出 |
| CTRL-E | 向上滚动一行 |
| CTRL-Y | 向下滚动一行 |

":s" 命令中的 "from" 部分实际上是一个 "匹配模式" (还记得吗？这是我们前面给 pattern 起的名字译者)，这与查找命令一样。例如， 要替换行首的 "the" 可以这样写:

```
:s/^the/these/
```

如果你要在 "from" 或者 "to" 中使用正斜杠，你需要在前面加上一个反斜杠。更简单的方法是用加号代替正斜杠。例如:

```
:s+one/two+one or two+
```

---

## *10.3* 命令范围

":substitute" 命令和很多其它的 ":" 命令一样，可以作用于选中的一些行。这称为一个 "范围"。

最简单的范围表达形式是 "{number},{number}"。例如:

```
:1,5s/this/that/g
```

这会在 1 到 5 行上执行替换命令。(包括第 5 行)。"范围" 总是放在一个命令的前面。

如果只用一个数值，表示某个指定的行:

```
:54s/President/Fool/
```

有些命令在不指定范围的时候作用于整个文件。要让它只作用于当前行可以用当前行范围标识 ".". ":write" 命令就是这样: 不指定范围的时候，它写入整个文件，如果要仅写入当前行，可以这样:

```
:.write otherfile
```

文件的第一行行号总是 1，最后一行又是多少呢？"$" 字符用于解决这个问题。例如，要修改当前行到文件末的全部内容，可以这样:

```
:.,$s/yes/no/
```

我们前面使用的 "%" 就是 "1,$" 的缩写形式，表示从文件首到文件末。

## 在范围中使用模式

假设你正在编辑一本书中的一章，并且想把所有的 "grey" 修改成 "gray"。但你只想修改这一章，不想影响其它的章节。另外，你知道每章的开头的标志是行首的单词为 "Chapter"。下面的命令会对你有帮助:

```
:?^Chapter?,/^Chapter/s=grey=gray=g
```

你可以看到这里使用了两个查找命令。第一个是 "?^Chapter?"，用于查找前一个行首的 "Chapter"，就是说 "?pattern?" 用于向前查找。同样，"/^Chapter/" 用于向后查找下一章。

为了避免斜杠使用的混淆，在这种情况下，"=" 字符用于代替斜杠。使用斜杠或使用其它字符其实也是可以的。

## 加减号

上面的方案其实还是有问题的: 如果下一章的标题行中包括 "grey"，这个 "grey" 也会被替换掉。如果你正好想这样就最好，可是正好你不想呢？这个时候你需要指定一个偏移。

要查找一个模式，并且使用它的前一行，需要这样:

```
/Chapter/-1
```

你可以用任意数值代替命令中的 1。要定位匹配点下的第二行，要这样:

```
/Chapter/+2
```

偏移还可以用于其它范围指定符。看一下下面这个例子:

```
:.+3,$-5
```

这指定当前行下面第三行到文件末倒数第五行的范围。

## 使用标记

除了指定行号，(这需要记住并把它敲出来)，你还可以使用标记。

在前面的例子中，你可以用标记指出第三章的位置。例如，用 "mt" 标记开头，再用 "mb" 标记结尾。然后你就可以用标记表示一个范围 (包括标记的那一行):

```
:'t,'b
```

可 视 模 式 和 范 围

你可以在可视模式中选中一些行。如果你现在输入 ":" 启动冒号命令模式，你会看到:

```
:'<,'>
```

现在，你可以输入剩下的命令，这个命令的作用范围就是可视模式中指定的范围。

> 备注:
> 如果使用可视模式选中行的一部分，或者用 CTRL-V 选中一个文本列块，然后执行冒号命令，命令仍作用于整行，而不只是选中的范围。这可能会在以后的版本中修正。

'< 和 '> 实际上是标记，分别标识可视模式的开始和结尾。这个标记一直有效，直到选中了其它的范围为止。你还可以用标记跳转命令 "'<" 跳转到选中文本的开始处。你还可以把这个标记和其它标记混合，例如:

```
:'>,$
```

这表示从选中部分的结尾到文件末。

指 定 行 数

如果你知道要修改多少行，你可以先输入一个数值再输入冒号。例如，如果你输入 "5:"，你会得到:

```
:.,.+4
```

现在你可以继续你的命令，这个命令将作用于当前行及其后 4 行。

---

## *10.4* global 命令

":global" 命令是 Vim 中一个更强大的命令（之一）。它允许你找到一个匹配点并且在那里执行一个命令。它的一般形式是:

```
:[range]global/{pattern}/{command}
```

这有点象 ":substitute" 命令。只是它不替换文本，而是执行 {command} 指定的命令。

> 备注:
> global 中执行的命令只能是冒号命令。普通模式命令不能在这里使用。如果需要，可以使用 |:normal| 命令。

假设你要把 "foobar" 修改为 "barfoo"，但只需要修改 C++ 风格的注释中的内容。这种注释以 "//" 开头。所以可以使用如下命令:

```
:g+//+s/foobar/barfoo/g
```

这个命令用 ":g" 开头，这是 ":global" 的缩写形式，就像 ":s" 是 ":substitute" 的缩写形式一样。然后是一个匹配模式，由于模式中包括正斜杠，我们用加号作分隔符，后面是一个把 "foobar" 替换成 "barfoo" 的替换命令。

全局命令的默认范围是整个文件，所以这个例子中没有指定范围。这一点与 ":substitute" 是不同的。后者只作用于一行。

这个命令并非完美。因为 "//" 可能出现在一行的中间，但替换命令会把前后的匹配点都替换了。

像 ":substitute" 一样，这里也可以使用各种各样的匹配模式。当你从后面的章节中学会更多的关于模式的知识，它们都可以用在这里。

---

## *10.5* 可视列块模式

CTRL-V 命令可以选中一个矩形文本块。有几个命令是专门用来处理这个文本块的。

在可视列块模式中，"$" 命令有些特别。当最后一个移动命令是 "$" 时，整个可视列块将被扩展到每一行的行尾。这种状态在你使用垂直移动命令的时候一直被保持，直到你使用水平移动命令为止。就是说，用 "j" 命令会保持这种状态，而 "h" 会退出。

### 插 入 文 本

"I{string}<Esc>" 命令把 {string} 插到可视列块的每一行的左边。你用 CTRL-V 进入可视列块模式，然后移动光标定义一个列块。接着输入 I 进入插入模式，并随后输入文本。这时，你输入的内容只出现在第一行。

然后你输入 <Esc> 结束输入，刚才输入的字符串将神奇地出现在每一行的可视区的左边。例如:

```
include one
include two
include three
include four
```

把光标移到第一行 "one" 的 "o"上，输入 CTRL-V。然后用 "3j" 向下移动到 "four"。现在你选中了四行的一个方块。接着输入:

```
Imain.<Esc>
```

结果将是:

```
include main.one
include main.two
include main.three
include main.four
```

如果选中的块经过一个短行，并且这行没有任何内容包括在可视列块中，则新的文本不会被插入到该行中。例如，对于下面的例子，用可视列块选中第一和第三行的 "long"，这样第二行的文本将不会被包括在可视列块中:

```
This is a long line
short
Any other long line

     ^^^^ 用可视列块选中的部分
```

现在输入 "Ivery <esc>"。结果将是:

```
This is a very long line
short
Any other very long line
```

可以注意到，第二行中没有插入任何文本。

如果插入的文本中包括一个新行，则 "I" 命令的效果与普通插入语句一样，只影响块的第一行。

"A" 命令的效果与 "I" 命令一样，只是把文字插入可视列块的右边，而且在短行中会插入文字。这样，你有在短行中插入文字与否的不同选择。

"A" 在如下情况会有一些特别: 选中一个可视列块然后用 "$" 命令使可视列块扩展到行尾。然后用 "A" 命令插入文本，文件将被插入到 "每一行" 的行尾。

还是用上面的例子，在选中可视列块后输入 "$A XXX<Esc>"，结果将是:

```
This is a long line XXX
short XXX
Any other long line XXX
```

出现这个效果完全是 "$" 命令的作用，Vim 能记住这个命令，如果你用移动命令选中相同的可视列块，是不会有这样的效果的。

## 修 改 文 本

可视列块中的 "c" 命令会删除整个可视列块并转入 "插入" 模式，使你可以开始文本，这些文本会被插入可视列块经过的每一行。

在上面的例子中，如果仍选中包括所有 "long" 的一个可视列块，然后输入 "c_LONG_<Esc>"，结果会变成:

```
This is a _LONG_ line
short
Any other _LONG_ line
```

与 "I" 命令一样，短行不会发生变化。而且在插入的过程中，你不能断行。

"C" 命令从块的左边界开始删除所有行的后半段，然后状态切换到 "插入" 模式让你输入文本。新的文本被插入到每一行的末尾。

在上面的例子中，如果命令改为 "Cnew text<Esc>"，你将获得这样的结果:

```
This is a new text
short
Any other new text
```

可以注意到，尽管只有 "long" 被选中，它后面的内容也被删除了。所以在这种情况下，块的左边界才是有意义的。

同样，没有包括在块中的行不会受影响。

还有一些命令只影响被选中的字符:

| | | |
|---|---|---|
| ~ | 交换大小写 | (a -> A 而 A -> a) |
| U | 转换成大写 | (a -> A 而 A -> A) |
| u | 转换成小写 | (a -> a 而 A -> a) |

以 一 个 字 符 填 充

要以某一个字符完全填充整个块，可以使用 "r" 命令。再次选中上例中的文本，然后键入 "rx":

```
This is a xxxx line
short
Any other xxxx line
```

备注:
如果你要在可视列块中包括行尾之后的字符，请参考 25 章的 'virtualedit' 特性。

平 移

"＞" 命令把选中的文档向右移动一个 "平移单位"，中间用空白填充。平移的起始点是可视列块的左边界。

还是用上面的例子，"＞" 命令会导致如下结果：

```
This is a         long line
short
Any other         long line
```

平移的距离由 'shiftwidth' 选项定义。例如，要每次平移 4 个空格，可以用这个命令：

```
:set shiftwidth=4
```

"＜" 命令向左移动一个 "平移单位"，但能移动的距离是有限的，因为它左边的不是空白字符的字符会挡住它，这时它移到尽头就不再移动。

连 接 若 干 行

"J" 命令连接被选中的行。其实就是删除所有的换行符。其实不只是换行符，行前后的多余空白字符会一起被删除而全部用一个空格取代。如果行尾刚好是句尾，就插入两个空格（参见 'joinspaces' 选项）

还是用那个我们已经非常熟悉的例子，这回的结果将是：

```
This is a long line short Any other long line
```

"J" 命令其实不关心选中了哪些字符，只关心块涉及到哪些行。所以可视列块的效果与 "v" 和 "V" 的效果是完全一样的。

如果你不想改变那些空白字符，可以使用 "gJ" 命令。

---

## *10.6* 读、写文件的一部分

当你在写一封 e-mail，你可能想包括另一个文件。这可以通过 ":read {filename}" 命令达到目的。这些文本将被插入到光标的下面。

我们用下面的文本作试验：

```
Hi John,
Here is the diff that fixes the bug:
Bye, Pierre.
```

把光标移到第二行然后输入:

```
:read patch
```

名叫 "patch" 的文件将被插入，成为下面这个样子:

```
Hi John,
Here is the diff that fixes the bug:
2c2
<       for (i = 0; i <= length; ++i)
---
>       for (i = 0; i < length; ++i)
Bye, Pierre.
```

":read" 支持范围前缀。文件将被插入到范围指定的最后一行的下面。所以 ":$r patch" 会把 "patch" 文件插入到当前文件的最后。

如果要插入到文件的最前面怎么办？你可以把文本插入到第 0 行，这一行实际上是不存在的。在普通的命令的范围中如果你用这个行号会出错，但在 "read" 命令中就可以:

```
:0read patch
```

这个命令把 "patch" 文件插入到全文的最前面。

## 保 存 部 分 行

要把一部分行写入到文件，可以使用 ":write" 命令。在没有指定范围的时候它写入全文，而指定范围的时候它只写入范围指定的行:

```
:.,$write tempo
```

这个命令写入当前位置到文件末的全部行到文件 "tempo" 中。如果这个文件已经存在，你会被提示错误。Vim 不会让你直接写入到一个已存在的文件。如果你知道你在干什么而且确实想这样做，就加一个叹号:

```
:.,$write! tempo
```

小 心: "!" 必须紧跟着 ":write"，中间不能留有空格。否则这将变成一个过滤器命令，这种命令我们在本章的后面会介绍。

## 添 加 内 容 到 文 件 中

本章开始的时候介绍了怎样把文本添加到寄存器中。你可以对文件作同样的操作。例如，把当前行写入文件：

```
:.write collection
```

然后移到下一个位置，输入：

```
:.write >>collection
```

"`>>`" 通知 Vim 把内容添加到文件 "collection" 的后面。你可以重复这个操作，直到获得全部你需要收集的文本。

---

## *10.7* 排版文本

在你输入纯文本时，自动换行自然会是比较吸引的功能。要实现这个功能，可以设置 'textwidth' 选项：

```
:set textwidth=72
```

你可能还记得在示例 vimrc 文件中，这个命令被用于所有的文本文件。所以如果你使用的是那个配置文件，实际上你已经设置这个选项了。检查一下该选项的值：

```
:set textwidth
```

现在每行达到 72 个字符就会自动换行。但如果你只是在行中间输入或者删除一些东西，这个功能就无效了。Vim 不会自动排版这些文本。

要让 Vim 排版当前的段落：

```
gqap
```

这个命令用 "gq" 开始，作为操作符，然后跟着 "ap"，作为文本对象，该对象表示 "一段" (a paragraph)。"一段" 与下一段的分割符是一个空行。

备注：
只包括空白字符的空白行不能分割 "一段"。这很不容易分辨。

除了用 "ap"，你还可以使用其它 "动作" 或者 "文本对象"。如果你的段落分割正确，你可以用下面命令排版整个文档：

```
gggqG
```

"gg" 跳转到第一行，"gq" 是排版操作符，而 "G" 是跳转到文尾的 "动作" 命令。

如果你没有清楚地区分段落。你可以只排版你手动选中的行。先移到你要格式化的行，执行 "gqj"。这会排版当前行和下面一行。如果当前行太短，下面一行会补上来，否则多余的部分会移到下面一行。现在你可以用 "." 命令重复这个操作，直到排版完所有的文本。

---

## *10.8* 改变大小写

你手头有一个分节标题全部是小写的。你想把全部 "section" 改成大写的。这可以用 "gU" 操作符。先在第一列执行:

```
                          gUw
section header        ---->       SECTION header
```

"gu" 的作用正好相反:

```
                          guw
SECTION header        ---->       section header
```

你还可以用 "g~" 来交换大小写。所有这些命令都是操作符，所以它们可以用于 "动作" 命令，文本对象和可视模式。

要让一个操作符作用于当前行，可以执行这个操作符两次。例如，"d" 是删除操作符，所以删除一行就是 "dd"。相似地，"gugu" 使整一行变成小写。这可以缩成 "guu"。"gUgU" 可以缩成 "gUU" 而 "g~g~" 则是 "g~~"。例如:

```
                                g~~
Some GIRLS have Fun     ---->   sOME girls HAVE fUN
```

---

## *10.9* 使用外部程序

Vim 有一套功能非常强大的命令，可以完成所有功能。但有些东西外部命令能够完成得更好或者更快。

命令 "!{motion}{program}" 用一个外部程序对一个文本块进行过滤。换句话说，它用一个文本块作为输入，执行一个由 {program} 指定的外部命令，然后用该程序的输出替代选中的文本块。

如果你不熟悉 UNIX 的过滤程序，上面的描述可以说是比较糟糕的。我们这里举个例子来说明一下。sort 命令能对一个文件排序。如果你执行下面的命令，未排序的文件 input.txt 会被排序并写入 output.txt。(这在 UNIX 和 Microsoft Windows 上都有效)

```
sort <input.txt >output.txt
```

现在在 Vim 中完成相同的功能。假设你要对 1 到 5 行排序。你可以先把光标定位在第一行，然后你执行下面的命令:

```
!5G
```

"!" 告诉 Vim 你正在执行一个过滤操作。然后 Vim 编辑器等待一个 "动作" 命令来告诉它要过滤哪部分文本。"5G" 命令告诉 Vim 移到第 5 行。于是，Vim 知道要处理的是第 1 行（当前行）到第 5 行间的内容。

由于在执行一个过滤命令，光标被 Vim 移到了屏幕的底部，并显示一个 "!" 作提示符。现在你可以输入过滤程序的名字，在本例中就是 "sort" 了。因此，你整个命令将是:

```
!5Gsort<Enter>
```

这个命令的结果是 sort 程序用前 5 行作为输入执行，程序的输出替换了原来的 5 行。

```
line 55                     line 11
line 33                     line 22
line 11          -->        line 33
line 22                     line 44
line 44                     line 55
last line                   last line
```

"!!" 命令用于对当前行执行过滤命令。在 Unix 上，"date" 命令能打印当前的时间和日期，所以，"!!date<Enter>" 用 "date" 的输出代替当前行。这在为文件加入时间戳的时候非常有用。

## 如果命令不执行怎么办

启动一个 shell，发送一个命令并捕获它的输出，这需要 Vim 知道这个 shell 程序是怎么工作的。如果你要使用过滤程序，你最好需要检查一下下面的选项:

| | |
|---|---|
| 'shell' | 指定 Vim 用于执行外部命令的 shell。 |
| 'shellcmdflag' | 传给 shell 的参数 |
| 'shellquote' | shell 程序使用的引号（用于引用命令） |
| 'shellxquote' | 用于命令和重定向文件名的引号 |
| 'shelltype' | shell 程序的类型（仅用于 Amiga） |
| 'shellslash' | 在命令中使用正斜杠（仅用于 MS-Windows 和相容系统） |
| 'shellredir' | 用于把命令输出写入文件所使用的字符串 |

在 Unix 上，这几乎不是问题。因为总共只有两种 shell 程序: "sh" 类的和 "csh" 类的。Vim 会检查选项 'shell'，并根据它的类型自动设置这些参数。

但在 MS-Windows 上，有很多不同的 shell 程序，所以你必须修改这些 shell 程序

以便过滤功能正常执行。详细情况请参考相应选项的帮助。

## 读入一个命令的输出

要把当前目录的内容读进文件，可以用如下命令:

Unix 上:

```
:read !ls
```

MS-Windows 上:

```
:read !dir
```

"ls" 或者 "dir" 的输出会被捕获并插入到光标下面。这好像读入一个文件一样，但是需要加上一个 "!" 让 Vim 知道后面是一个命令。

这些命令还可以带参数。而且前面还可以带一个范围用于告诉 Vim 把这行放在什么地方:

```
:0read !date -u
```

这将用 UTC 格式把当前的时间插入到文件开头。(当然了，你的 date 命令必须能够接受 -u 选项。) 注意 这与 "!!date" 的区别: "!!date" 替代一行，而 ":read !date" 插入一行。

## 把文本输出到一个命令

Unix 命令 "wc" 用于统计单词数目。要统计当前文件有多少个单词，可以这样:

```
:write !wc
```

这和前面的写入命令一样，但文件名前面改为一个 "!" 用于告诉 Vim 后面是一个要被执行的外部命令。被写入的文本将作为指定命令的标准输入。这个输出将是:

```
       4      47     249
```

"wc" 命令惜字如金。这表示你有 4 行，47 个单词和 249 个字符。

注意不要错写成:

```
:write! wc
```

这会强制把当前文件存到当前目录的 "wc" 文件中。在这里空格的位置是非常重要的!

---

重 画 屏 幕

如果外部程序产生一个错误信息，屏幕显示就会乱掉。Vim 颇重效率，所以它只刷新那些需要刷新的地方。可是它不可能知道其它程序修改了哪些地方。要强制 Vim 重画整个屏幕:

```
CTRL-L
```

从崩溃中恢复

你的计算机崩溃过吗？是不是还正好在你编辑了几个小时以后？不要惊慌！Vim 已经保存了大部分的信息使你可以恢复你的大多数数据。本章告诉你怎样恢复这些数据并向你介绍 Vim 是如何处理交换文件的。




下一章: |usr_12.txt| 小窍门
前一章: |usr_10.txt| 做大修改
目录: |usr_toc.txt|


---

## *11.1* 基本恢复

在大多数情况下，恢复一个文件相当简单。假设你知道正在编辑的是哪个文件（并且硬盘没坏的话）。可以用 "-r" 选项启动 Vim:

```
vim -r help.txt
```

Vim 会读取交换文件（用于保存你的编辑数据的文件）并且提取原文的编辑碎片。如果 Vim 恢复了你的改变，你会看到如下文字（当然了，文件名会不一样）:

```
Using swap file ".help.txt.swp"
Original file "~/vim/runtime/doc/help.txt"
Recovery completed. You should check if everything is OK.
(You might want to write out this file under another name
and run diff with the original file to check for changes)
You may want to delete the .swp file now.
```

(译者注: 中文情况下是:

```
使用交换文件 ".help.txt.swp"
原文件 "~/vim/runtime/doc/help.txt"
恢复完成。请确定一切正常。
(你可能想要把这个文件另存为别的文件名，
再执行 diff 与原文件比较以检查是否有改变)
现在可以删除 .swp 文件。
)
```

为了安全起见，可以用另一个文件名保存这个文件：

```
:write help.txt.recovered
```

可以把这个文件与原文件作一下比较，看看恢复的效果如何。这方面 Vimdiff 可以帮很大的忙（参见 |08.7|）。例如：

```
:write help.txt.recovered
:edit #
:diffsp help.txt
```

注意用一个比较新的原文件来比较（你在计算机崩溃前最后保存过的文件），并且检查有没有东西丢失了（由于某些问题导致 Vim 无法恢复）。

如果在恢复的过程中 Vim 显示出一些警告信息，注意小心阅读。这应该是很少见的。

如果恢复产生的文件和文件内容完全一致，你会看到以下消息：

```
Using swap file ".help.txt.swp"
Original file "~/vim/runtime/doc/help.txt"
Recovery completed. Buffer contents equals file contents.
You may want to delete the .swp file now.
```

通常这是因为你已经恢复过改变，或者修改后写入了文件。此时删除交换文件应该安全。

最后所做的一些修改不能恢复是正常的。Vim 在你停止大约 4 秒不输入的时候或者输入大约两百个字符以后才会更新交换文件。这间可以通过 'updatetime' 和 'updatecount' 两个选项来调整。这样，如果系统崩溃前 Vim 没有更新交换文件，最后一次更新后编辑的内容就会丢失。

如果你编辑的时候没有给定文件名，可以用一个空的字符串来表示文件名：

```
vim -r ""
```

你需要进入原来的目录执行这个命令，否则 Vim 是找不到这个交换文件的。

## *11.2*  交换文件在哪

Vim 可以把交换文件保存在几个不同的地方。通常是原文件所在的目录。要知道这一点，进入该目录，然后输入:

```
vim -r
```

Vim 会列出所有它能找到的交换文件。它还会从其它目录寻找本目录文件的交换文件，但它不会寻找其它目录里的交换文件，更不会遍及整个目录树。

这个命令的输出如下:

```
Swap files found:
   In current directory:
1.    .main.c.swp
          owned by: mool   dated: Tue May 29 21:00:25 2001
         file name: ~mool/vim/vim6/src/main.c
          modified: YES
         user name: mool   host name: masaka.moolenaar.net
        process ID: 12525
   In directory ~/tmp:
      -- none --
   In directory /var/tmp:
      -- none --
   In directory /tmp:
      -- none --
```

(译者: 中文的情形如下:

```
找到以下交换文件:
   位于当前目录:
1.    .main.c.swp
            所有者: mool    日期: Tue May 29 21:00:25 2001
            文件名: ~mool/vim/vim6/src/main.c
            修改过: 是
            用户名: mool     主机名: masaka.moolenaar.net
           进程 ID: 12525
   位于目录 ~/tmp:
        -- 无 --
   位于目录 /var/tmp:
        -- 无 --
   位于目录 /tmp:
        -- 无 --
)
```

如果有几个交换文件，其中一个可能是你要的，Vim 会给出一个文件列表，你需要输入一个表示这个文件的数字。小心检查那几个文件的时间，并确定哪一个才是你需要的。
万一你不知道是哪个的话，一个一个试一试。

使 用 指 定 的 交 换 文 件

如果你知道要用哪个文件，你可以指定交换文件的名字。Vim 会找出交换文件所对应的原始文件的名字。

例如:

```
Vim -r .help.txt.swo
```

这个方法在交换文件在一个非预期的目录中时很有用。Vim 知道 *.s[uvw][a-z] 模式的文件是交换文件。

如果这还不行，看看 Vim 报告的文件名是什么，然后根据需要给文件换名。根据 'directory' 选项的值你可以知道 Vim 会把交换文件放到什么地方。

> 备注:
> Vim 在 'dir' 选项指定的目录中寻找名为 "filename.sw?" 的交换文件。如果通配符不能正常工作（例如 'shell' 选项不正确），Vim 转而尝试文件 "filename.swp"。如果仍失败，你就只能通过给定交换文件的名称来恢复原来的文件了。

---

## *11.3* 是不是崩溃了？ *ATTENTION* *E325*

Vim 尽可能保护你不要做傻事。有时你打开一个文件，天真地以为文件的内容会显示出来。可是，Vim 却给出一段很长的信息:

```
E325: ATTENTION
Found a swap file by the name ".main.c.swp"
          owned by: mool   dated: Tue May 29 21:09:28 2001
         file name: ~mool/vim/vim6/src/main.c
          modified: no
         user name: mool   host name: masaka.moolenaar.net
        process ID: 12559 (still running)
While opening file "main.c"
             dated: Tue May 29 19:46:12 2001

(1) Another program may be editing the same file.
    If this is the case, be careful not to end up with two
    different instances of the same file when making changes.
```

```
    Quit, or continue with caution.

(2) An edit session for this file crashed.
    If this is the case, use ":recover" or "vim -r main.c"
    to recover the changes (see ":help recovery").
    If you did this already, delete the swap file ".main.c.swp"
    to avoid this message.
```

(译者注：翻译成中文如下：

```
        E325: 注意
发现交换文件 "main.c.swp"
     所有者: mool      日期: 2001 年 5 月 29 日 星期二 21:09:28
     文件名: ~mool/vim/vim6/src/main.c
     修改过: 否
     用户名: mool    主机名: masaka.moolenaar.net
     进程号: 12559 (仍在运行)
正在打开文件 "main.c"
       日期: 2001 年 5 月 29 日 星期二 19:46:12

(1) 另一个程序可能也在编辑同一个文件。
    如果是这种情况，修改时请注意避免同一个文件产生两个不同的版本。

    退出，或小心地继续。

(2) 上次编辑此文件时崩溃。
    如果是这样，请用 ":recover" 或 "vim -r main.c"
    恢复修改的内容 (请见 ":help recovery")。
    如果你已经进行了恢复，请删除交换文件 ".main.c.swp"
    以避免再看到此消息。
```

)

你遇到这个信息是因为 Vim 发现你编辑的文件的交换文件已经存在。这一定是有什么地方出问题了。可能的原因有两个：

1. 这个文件正在被另一个进程编辑。注意有 "process ID" 那行。它看起来是这样的：

```
process ID: 12559 (still running)
```

"still running" 表示同一台计算机上有一个进程正在编辑这个文件。在非 Unix 的系统上你不会得到这个信息。而如果你通过网络编辑这个文件，可能也得不到这个信息，因为那个进程不在你的机器上。在这两种情况下，你要自己找到原因。

如果另一个 Vim 正在编辑这个文件，继续编辑会导致同一个文件有两个版本。最

后存盘的文件会覆盖前一个版本。这样的结果是一些编辑数据丢失了。这种情况下，你最好退出这个 Vim。

2. 交换文件可能是由于前一次 Vim 或者计算机崩溃导致的。检查提示信息中的日期。如果交换文件比你正在编辑的文件新，而且出现这个信息:

```
modified: YES
```

这就表明你很可能需要恢复了。

如果文件的日期比交换文件新，可能是在崩溃后被修改过了（也许你已经恢复过，只是没有删除交换文件？），也可能文件在崩溃前保存过，但这发生在在最后一次写入该交换文件之后（那你运气了，你根本不需要这个旧的交换文件）。Vim 会用如下语句提醒你:

```
NEWER than swap file!
```

（译者注: 意为 "文件比交换文件新"）

## 无 法 读 取 的 交 换 文 件

有时下面这样的信息

```
[cannot be read]
```

或［无法读取］（中文信息，译者）

会出现在交换文件的文件名之下。这可好可坏，依情况而定。

如果上次编辑在作出任何修改前就崩溃了的话，是好事。这样交换文件的长度为 0。你只要删除之然后继续即可。

如果情况是你对交换文件没有读权限，就比较糟糕。你可能得以只读方式浏览该文件。或者退出。在多用户系统中，如果你以别人的身份登录并做了上一次修改，先退出登录然后以那个身份重新登录可能会 "治愈" 该读取错误。不然的话，你得找出是谁做的上一次修改（或正在修改），然后和那个人聊聊。

如果情况是交换文件所在的磁盘物理性地损坏了，就非常糟糕了。幸运的是，这种情况几乎不会发生。
你可能需要以只读方式查看文件（如果允许的话），看看到底有多少改动被 "忘记" 了。
如果你是改动文件的那个人，准备好重做你的改动。

## 怎 么 办？ *swap-exists-choices*

如果 Vim 版本支持对话框，你可以从对话框的五个选择中（译者注：原文如此）挑一个：

```
Swap file ".main.c.swp" already exists!
[O]pen Read-Only, (E)dit anyway, (R)ecover, (Q)uit, (A)bort, (D)elete it:
```

（译者：含义是：
交换文件 ".main.c.swp" 已经存在！
以只读方式打开([O])，直接编辑((E))，恢复((R))，退出((Q))，中止((A))，删除交换文件((D))：
）

O　用只读方式打开文件。当你只是想看看文件的内容，而不打算恢复它的时候用这个选项。你可能知道有人在编辑它，但你想看看它的内容，而不会修改它。

E　直接编辑。小心使用这个选择！如果这个文件已经被另一个文件打开，你编辑它会导致它有两个版本。Vim 已经警告过你了，安全比事后说对不起要好。

R　从交换文件中恢复文件。如果你知道交换文件中有新的东西，而你想恢复它，选择这一项。

Q　退出。不再编辑该文件。在有另一个 Vim 编辑该文件的时候选这一项。
　　如果你刚打开 Vim，这会退出 Vim。当你用多个窗口打开几个文件，Vim 只会在第一个文件遇到交换文件的时候退出。如果你是通过编辑命令打开文件，该文件不会被载入，Vim 会回到原来的文件中。

A　中止。类似 (Q) 退出，但同时中止更多的命令。这在试图加载一个编辑多个文件的脚本（例如一个多窗口的会话）时很有用。

D　删除交换文件。当你能确定你不再需要它的时候选这一项。例如，它不包括修改的数据，或者你的文件比交换文件新。
　　在 Unix 系统上，只有建立这个交换文件的进程不再运行，这个选择才会出现。

如果没有出现对话框（你使用的 Vim 不支持对话框），你只能手工处理。要恢复一个文件，使用如下命令：

```
:recover
```

Vim 不是总能检测到一个文件有交换文件的。当另一个会话把交换文件放到别的位置或者在编辑另一台机器的文件的时候，双方使用的交换文件路径不一样都会发生这个问题。所以，不要老是等 Vim 来提醒你。

如果你确实不想看到这个信息，你可以在 'shortmess' 选项中加上 'A' 标志位。不过一般你不需要这样做。

关于加密和交换文件关系的注释，见 |:recover-crypt|。

小窍门

通过组合一些命令，你可以用 Vim 完成几乎所有的工作。本章将介绍一些有用的命令组合。涉及的命令大都是前面章节介绍过的，但也会有一点新命令。




下一章: |usr_20.txt| 快速键入命令行命令
前一章: |usr_11.txt| 从崩溃中恢复
目录: |usr_toc.txt|


---

## *12.1* 单词替换

替换命令可以在全文中用一个单词替换另一个单词:

```
:%s/four/4/g
```

"%" 范围前缀表示在所有行中执行替换。最后的 "g" 标记表示替换行中的所有匹配点。

如果你有一个象 "thirtyfour" 这样的单词，上面的命令会出错。这种情况下，这个单词会被替换成 "thirty4"。要解决这个问题，用 "\<" 来指定匹配单词开头:

```
:%s/\<four/4/g
```

显然，这样在处理 "fourteen" 的时候还是会出错。用 "\>" 来解决这个问题:

```
:%s/\<four\>/4/g
```

如果你在编码，你可能只想替换注释中的 "four"，而保留代码中的。由于这很难指定，可以在替换命令中加一个 "c" 标记，这样，Vim 会在每次替换前提示你:

```
:%s/\<four\>/4/gc
```

在 多 个 文 件 中 替 换

假设你需要替换多个文件中的单词。你的一个选择是打开每一个文件并手工修改。另外，如果使用 "记录－回放" 命令会更快。

假设你有一个包括有 C++ 文件的目录，所有的文件都以 ".cpp" 结尾。有一个叫 "GetResp" 的函数，你需要把它改名为 "GetAnswer"。

| | |
|---|---|
| vim *.cpp | 启动 Vim，用当前目录的所有 C++ 文件作为文件参数。启动后你会停在第一个文件上。 |
| qq | 用 q 作为寄存器启动一次记录。 |
| :%s/\<GetResp\>/GetAnswer/g | 在第一个文件中执行替换。 |
| :wnext | 保存文件并移到下一个文件。 |
| q | 中止记录。 |
| @q | 回放 q 中的记录。这会执行又一次替换和 ":wnext"。你现在可以检查一下记录有没有错。 |
| 999@q | 对剩下的文件执行 q 中的命令 |

Vim 会在最后一个文件上报错，因为 ":wnext" 无法移到下一个文件上。这时所有的文件中的操作都完成了。

> 备注:
> 在回放记录的时候，任何错误都会中止回放的过程。所以，要注意保证记录中的命令不会产生错误。

这里有一个陷阱: 如果有一个文件不包含 "GetResp"，Vim 会报错，而整个过程会中止，要避免这个问题，可以在替换命令后面加一个标记:

```
:%s/\<GetResp\>/GetAnswer/ge
```

"e" 标记通知 ":substitute" 命令找不到不是错误。

---

## *12.2* 把 "Last, First" 改成 "First Last"

你有如下样式的一个名字列表:

```
Doe, John
Smith, Peter
```

你想把它改成:

```
John Doe
Peter Smith
```

这可以用一个命令完成:

```
:%s/\([^,]*\), \(.*\)/\2 \1/
```

我们把这个命令分解成几个部分。首先，很明显它是一个替换命令。"%" 是行范围，表示作用于全文。这样替换命令会作用于全文的每一行。

替换命令的参数格式是 "from/to"，正斜杠区分 "from" 模式和 "to" 字符串。所以，"from" 部分是:

```
                                                \([^,]*\), \(.*\)

第一对 \( 和 \) 之间的部分匹配 "Last"            \(     \)
    匹配除逗号外的任何东西                         [^,]
    任意多次                                        *
匹配逗号                                              ,
第二对 \( 和 \) 之间的部分匹配 "First"                  \(  \)
    匹配任意字符                                       .
    任意多次                                            *
```

在 "to" 部分，我们有 "\2" 和 "\1"。这些称为 "反向引用"。它们指向前面模式中的 \( 和 \) 间的部分。"\2" 指向模式中的第二对 \( 和 \) 间的部分，也就是 "First" 名 (译者注: 英文中 Last Name 表示姓，即家族名，后面的 First Name 表示名字)。"\1" 指向第一对 \( \)，即 "Last" 名。

你可以在替换部分使用多达 9 个反向引用。"\0" 表示整个匹配部分。还有一些特殊的项可以用在替换命令中。请参阅 |sub-replace-special|。

---

## *12.3* 排序

在你的 Makefile 中常常会有文件列表。例如:

```
OBJS = \
        version.o \
        pch.o \
        getopt.o \
        util.o \
        getopt1.o \
        inp.o \
        patch.o \
        backup.o
```

要对这个文件列表排序可以用一个外部过滤命令:

```
/^OBJS
j
:.,/^$/-1!sort
```

这会先移到 "OBJS" 开头的行，向下移动一行，然后一行行执行过滤，直到遇到一个空行。你也可以先选中所有需要排序的行，然后执行 "!sort"。那更容易一些，但如果有很多行就比较麻烦。

上面操作的结果将是:

```
OBJS = \
        backup.o
        getopt.o \
        getopt1.o \
        inp.o \
        patch.o \
        pch.o \
        util.o \
        version.o \
```

注意，列表中每一行都有一个续行符，但排序后就错掉了！"backup.o" 在列表的最后，不需要续行符，但排序后它被移动了。这时它需要有一个续行符。

最简单的解决方案是用 "A \<Esc>" 补一个续行符。你也可以在最后一行放一个续行符，由于后面有一个空行，这样做是不会有问题的。

---

## *12.4* 反转行顺序

|:global| 命令可以和 |:move| 命令联用，将所有行移动到文件首部。结果是文件被按行反转了次序。命令是:

```
:global/^/m 0
```

缩写:

```
:g/^/m 0
```

正则表达式 "^" 匹配行首（即使该行是一个空行）。|:move| 命令将匹配的行移动到那个神秘的第 0 行之后。这样匹配的行就成了文件中的第一行。由于 |:global| 命令不会被改变了的行号搞混，该命令继续匹配文件中剩余的行并将它们一一变为首行。

这对一个行范围同样有效。先移动到第一行上方并做标记 't (mt)。然后移动到范围的最后一行并键入:

```
:'t+1,.g/^/m 't
```

---

## *12.5*  单词统计

有时你要写一些有最高字数限制的文字。Vim 可以帮你计算字数。

如果你需要统计的是整个文件的字数，可以用这个命令：

```
g CTRL-G
```

不要在 "g" 后面输入一个空格，这里只是方便阅读。

它的输出是：

```
Col 1 of 0; Line 141 of 157; Word 748 of 774; Byte 4489 of 4976
```

(译者注：中文是：
第 1/0 列；第 141/157 行；第 748/774 个词；第 4489/4976 个字节
)

你可以看到你在第几个单词（748）上以及文件中的单词总数（774）。

如果你要知道的是全文的一部分的字数，你可以移到该文本的开头，输入 "g CTRL-G"，然后移到该段文字的末尾，再输入 "g CTRL-G"，最后心算出结果来。这是一种很好的心算练习，不过不是那么容易。比较方便的办法是使用可视模式，选中你要计算字数的文本，然后输入 "g CTRL-G"，结果将是：

```
Selected 5 of 293 Lines; 70 of 1884 Words; 359 of 10928 Bytes
```

(译者注：中文是：
选择了 5/293 行；70/1884 个词；359/10928 个字节
)

要知道其它计算字数，行数和其它东西总数的方法，可以参见 |count-items|。

---

## *12.6*  查阅 man 信息  *find-manpage*

编辑一个脚本文件或者 C 程序的时候，有时你会需要从 man 手册中查询某个命令或者函数的用法（使用 Unix 的情况下）。让我们先用一个简单的方法：把鼠标移到对应的单词上然后输入：

```
K
```

Vim 会在对应的单词上执行外部命令：man。如果能找到相应的手册，那个手册页就会被显示出来。它常常用 more 一类的程序显示页面。在手册滚动到文件末并回车，控制就会回到 Vim 中。

这种方法的缺点是你不能同时查看手册和编辑文档。这里有一种办法可以把手册显示到一个 Vim 的窗口中。首先，加载 man 文件类型的外挂:

```
:runtime! ftplugin/man.vim
```

如果你经常用到这种方法，可以把这个命令加到你的 vimrc 文件中。现在你可以用 ":Man" 命令打开一个显示 man 手册的窗口了:

```
:Man csh
```

你可以在这个新的窗口中上下滚动，而手册的本文会用语法高亮的形式显示。这样，你可以找到需要的地方，并用 CTRL-W w 跳转到原来的窗口中继续工作。

要指定手册的章节，可以在手册名称前面指定。例如，要找第三章的 "echo":

```
:Man 3 echo
```

要跳转到另一个由 "word(1)" 形式定义的手册，只要在上面敲 CTRL-]。无论怎样，":Man" 命令总使用同一个窗口。

要显示当前光标下的单词的手册，这样:

```
\K
```

(如果你重定义了 <Leader>，用那个字符代替上面命令的反斜杠)。
例如，你想知道下面语句中的 "strstr()" 函数的返回值:

```
if ( strstr(input, "aap") == )
```

可以把光标移到 "strstr" 并输入 "\K"。手册使用的窗口会显示 strstr() 的信息。

---

## *12.7* 删除多余的空格

有些人认为行末的空格是无用，浪费而难看的。要删除这些每行后面多余的空格，可以执行如下命令:

```
:%s/\s\+$//
```

命令前面指明范围是 "%"，所以这会作用于整个文件。"substitute" 命令的匹配模式是 "\s\+$"。这表示行末 ($) 前的一个或者多个 (\+) 空格 (\s)。后面我们会介绍怎样写这样的模式。|usr_27.txt|。

替换命令的 "to" 部分是空的: "//"。这样就会删除那些匹配的空白字符。

另一种没有用的空格是 Tab 前面的字符。通常这可以删除而不影响格式。但并不是总这

样！所以，你最好手工删除它。执行如下命令：

```
/
```

你什么都看不见，其实这是一个空格加一个 TAB 键。相当于 "/<Space><Tab>"。现在，你可以用 "x" 删除多余的空格，并保证格式没有改变。接着你可以用 "n" 找到下一个位置并重复这个操作。

---

## *12.8* 查找单词的使用位置

如果你是一个 UNIX 用户，你可以用 Vim 和 grep 命令的组合来完成编辑包括特定单词的所有文件的工作。这在你编辑一个程序而且想查看和编辑看所有的包括使用某个变量的文件的时候非常有用。

举个例子，假设想编辑所有包括单词 "frame_counter" 的 C 源文件，你可以执行如下命令：

```
vim `grep -l frame_counter *.c`
```

让我们分析一下这个命令。grep 从一组文件中查找特定的单词。由于指定了 -l 参数，grep 只列出文件而不打印匹配点。被查找的单词是 "frame_counter"，其实这可以是任何正则表达式。(注意：grep 所使用的正则表达式与 Vim 使用的不完全一样)。

整个命令用反引号 (`) 包起来，这告诉 UNIX 的 shell 使用该命令的输出作为命令行的一部分。于是，grep 命令产生一个文件列表，并作为 Vim 的命令参数。Vim 将编辑 grep 列出来的所有文件。你可以通过 ":next" 和 ":first" 命令一个一个处理这些文件。

### 找到每一行

上面的命令只是找到包括单词的那个文件。你还需要知道单词在该文件中出现的地方。

Vim 有一个内置的命令用于在一组文件中找一个指定的字符串。例如，如果你想在所有的 C 文件中查找 "error_string"，可以使用如下命令：

```
:grep error_string *.c
```

这会使 Vim 在所有指定的文件 (*.c) 中查找 "error_string"。Vim 会打开第一个匹配的文件并将光标定位在第一个匹配行。要到下一个匹配行 (无论在哪个文件)，可以执行 "cnext" 命令。要回到上一个匹配行，可以用 ":cprev" 命令。使用 "clist" 可以看到所有的匹配点。

":grep" 命令会使用一个外部的程序。可能是 grep (在 Unix 上) 或者 findstr (在 Windows 上)。你可以通过 'grepprg' 选项修改这个设置。

---

如果你觉得这些，还不足你对 vim 了解，你可以到下面的站点：

http://vimcdoc.sourceforge.net/doc/help.html

VIM USER MANUAL - by Bram Moolenaar

目　　录　　*user-manual*

---

## 总览

初步知识



高效的编辑



调节 Vim

|usr_43.txt|  使用文件类型
|usr_44.txt|  自定义语法高亮
|usr_45.txt|  选择你的语言

让 Vim 工作
|usr_90.txt|  安装 Vim

参考手册
|reference_toc|  关于所有命令更详细的信息

本手册的 HTML 版本和 PDF 版本可以从以下这个地址得到:

http://vimdoc.sf.net （英文）
http://vimcdoc.sf.net （中文）