

vim 中文用户手册

目录

1. [Vim基础](#)
2. [移动](#)
3. [做小改动](#)
4. [使用语法高亮](#)
5. [分隔窗口](#)
6. [做大修改](#)
7. [小窍门](#)

vim 基础

1. [第一次运行vim](#)
2. [插入文本](#)
3. [移动光标](#)
4. [删除字符](#)
5. [撤销与重做](#)
6. [其它编辑命令](#)
7. [退出](#)
8. [寻求帮助](#)
9. [返回目录](#)

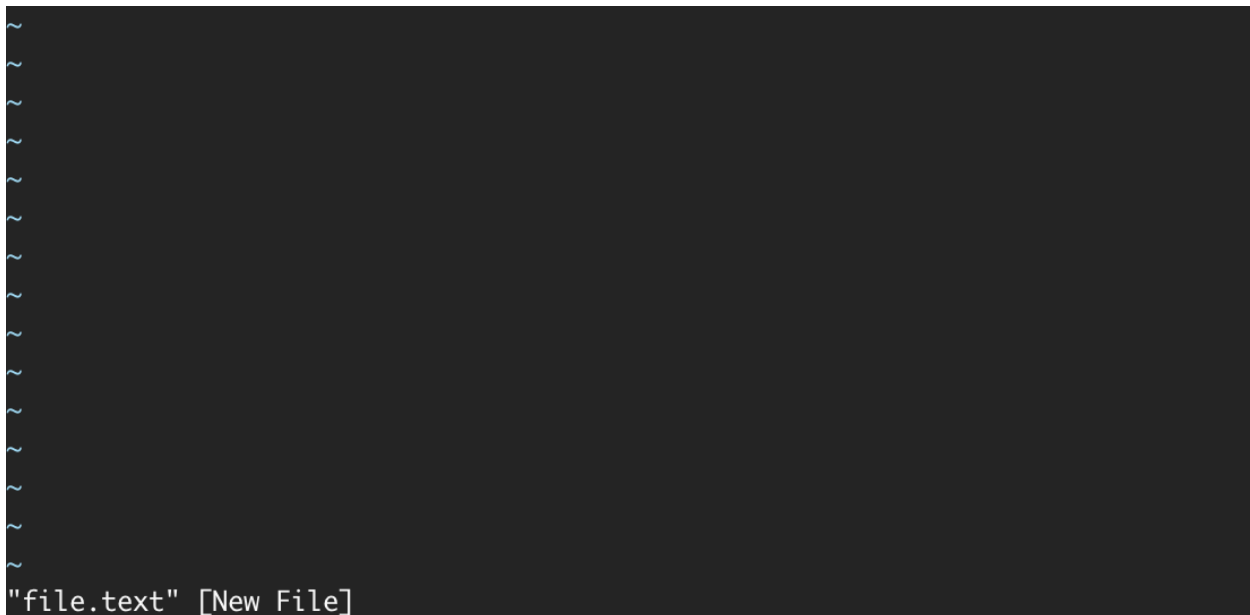
95后菜鸟码农，努力精进中，欢迎大家扫码加我好友共同学习成长，现在还可免费领取 10T的各类学习资料，同时可参与每日 免费包邮抽奖活动、 现金红包 等，真诚无套路！



第一次运行vim

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。屏幕看起来会像下面这样：



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。

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 菜单栏。后面会有更多的描述。

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` 是插入 (`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，你随后还能够修正它。

按 `<Esc>` 键退出插入模式而回到普通模式，如果不清楚当前处于什么模式，可以在命令行模式下输入以下命令查看：

```
: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--)，这表示你在插入模式。

移动光标

回到普通模式后，你可以使用如下命令移动光标:

按键	方向
h	左
j	下
k	上
l	右

删除字符

- 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)。

- 删除一整行

使用 `"dd"` 命令，后一行会移上来填掉留下的空行

- 删除一个换行符

在 Vim 中你可以把两行连起来，这意味着删除两行间的换行符。`"J"` 命令用于完成这个功能。以下面两行为例：

```
A young intelligent
turtle
```

把光标移到第一行，然后按 `"J":`

```
A young intelligent turtle
```

撤销与重做

- 撤销

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 输入7次 `x` 命令 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
```

注意: 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。要修正这个问题, 看看这里: [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最近编辑的行上的操作。输入该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
A very intelligent turtle
用 "U" 恢复行 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。

其它编辑命令

- 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, 附加) 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：

- 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。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。

- 指定计数

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>"。计数 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"。计数总是放在要被处理多次的命令的前面。

退出

- 退出

使用 "zz" 命令可以退出。这个命令保存文件并退出。

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；

- 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。

如果你关心细节，此命令有 三部分组成：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。

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！

如果你知道自己想要找什么，用帮助系统里查找通常比 `Google` 要方便。因为所有主题符合一定的风格指导。

帮助的另一优点是对于你特定的 `Vim` 系统。你不会看到之后加入的命令的帮助。这对你用不上。

要获得一般的帮助，用这个命令：

```
:help
```

要获得特定主题的帮助，使用如下命令：

```
:help {主题}
```

- 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'
```

- 下面有所有模式的前缀列表: `help-summary`
- 特殊键以尖括号包围。例如, 要找到关于插入模式的上箭头键的帮助, 用此命令:

```
:help i_<Up>
```

- 如果你看到一个你不能理解的错误信息, 你可以使用使用E开头的错误号找关于它的帮助:

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

```
:help E37
```

- 键入主题后用 `Ctrl-D` 让 Vim 显示所有的可用主题。也可按 `Tab` 来补全:

```
:help some<Tab>
```

- 关于如何使用 `help` 的详情:

```
:help helphelp
```

- 跟随竖杠之间的链接转到相关帮助。可从详细帮助转到用户文档，这里的一些命令解释更加贴近用户，而不过于繁琐。例如：

```
:help pattern.txt
```

- 选项以单引号包围。如要转到 `list` 选项的帮助主题：

```
:help 'list'
```

- 如果你只知道你想找某个选项，也可用：

```
:help options.txt
```

- 来打开描述所有选项处理的帮助页面，然后用正规表达式搜索，如 `textwidth`。若干选项有自己的命名空间，例如：

```
:help cpo-<letter>
```

- 可查找 `'cptions'` 设置的相关标志位，把 `<letter>` 替代为特定的标志位，如：

```
:help cpo-;
```

- 而要查 `guioption` 的标志位：

```
:help go-<letter>
```

- 普通模式命令没有前缀。如要转到 `"gt"` 命令的帮助页面：

```
:help gt
```

- 插入模式命令以 `i_` 开始。如关于删除单词的帮助：

```
:help i_CTRL-W
```

- 可视模式命令以 `v_` 开始。如跳转到可视区域另一边的帮助：

```
:help v_o
```

- 命令行编辑和参数以 `c_` 开始。如使用命令行参数 `%` 的帮助：

```
:help c_%
```

- Ex-命令总是以 `:"` 开始，如要转到 `:"s"` 命令的帮助：


```
:help :s
```

- 专门用于调试的命令以 ">" 开始。如要转到 "cont" 调试命令的帮助:

```
:help >cont
```

- 键组合。通常以指示要使用的模式的单个字母开始。例如:

```
:help i_CTRL-X
```

- 正规表达式项目总是以 "/" 开始。如要得到 Vim 正规表达式的 "\+" 量词的帮助:

```
:help /\+
```

- 如果你需要知道所有关于正规表达式的情况, 从这里开始:

```
:help pattern.txt
```

- 寄存器总是以 "quote" 开始。如要了解特殊的 ":" 寄存器:

```
:help quote:
```

- Vim 脚本可见,

```
:help eval.txt
```

- `:h expr-X` 描述语言的方方面面, 其中 "X" 是单个字母。如

```
:help expr-!
```

- 帮助页面 `:h map.txt` 讲到映射。用

```
:help mapmode-i
```

- 来查找 `:imap` 命令。另 `:map-topic` 可用来查找关于映射的特定子主题, 如:

```
:help :map-local
```

- `:h command-topic` 讲述命令的定义, 所以用

```
:help command-bar
```

- 高亮组。总是以 `hl-groupname` 开始。如

```
:help hl-WarningMsg
```

- 语法高亮使用命名空间 `:syn-topic`。如

```
:help :syn-conceal
```

移动

1. [词移动](#)
2. [移动到行首或行尾](#)
3. [移动到指定的字符](#)
4. [括号匹配](#)
5. [移动到指定的行](#)
6. [确定当前位置](#)
7. [滚屏](#)
8. [简单查找](#)
9. [简单的查找模式](#)
10. [使用标记](#)
11. [返回目录](#)

词移动

要移动光标向前跳一个词，可以使用 `"w"` 命令。像大多数 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单词的词首：

```
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 `"I"` 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 认为是单词组成部分 24 的字符，请参见 `'iskeyword'` 选项。如果你在此帮助文件里直接试验，先复位 `'iskeyword'`，此例才能工作：

```
:set iskeyword&
```

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。

`"^"` 命令把光标移动到一行的第一个非空字符，而 `"0"` 命令 (零) 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)
```

像大多数移动命令一样，`"$"` 命令接受计数前缀。但是 "移动到一行的行尾 n 次" 没有什么意义，所以它会使光标移动到另一行。例如，`"1$"` 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。

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"` 代表 `"Find"` (寻找)。例如，假定你在下行行首，而想移动到单词 `"human"` 的 h 那里。执行命令 `"fh"` 即可：

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

这个例子里同时演示 `"fy"` 命令移动到了 `"really"` 的词尾。你可以在这个命令前面加计数前缀，所以，你可以用 `"3f1"` 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" (到达)。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。备注：<Esc> 可以中止大部分命令，而不仅仅是查找。

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: 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， "%" 会先正向查找找到一个。比如当光标停留在上例中的

行首时， "%" 会正向查找到第一个 "("。再按一次会移动到它的匹配处。

```
if (a == (b * c) / d)
```

---+----->

%

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 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。加上一个计数前缀，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。）

如果没有计数前缀，"G" 命令把光标移动到文件末。移动到文件首的命令是 "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
    text text text text
    last line of a file
```

另一个定位行的方法是使用带计数前缀的 "%" 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。

提示："**H**" 表示 Home (头)，"**M**" 表示 Middle (中) 而 "**L**" 表示 Last (尾)。另外一种记法，"**H**" 表示 High (高)，"**M**" 表示 Middle (中) 而 "**L**" 表示 Low (低)。

确定当前的位置

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)：

```
"file.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. 置位 `'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，还有数值或者字符串类型的。在用到的时候会 给出一些例子的。

3. 置位 `'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。

滚屏

- `CTRL-U` 命令向下滚动半屏。想象一下通过一个视窗看着你的文本，然后把这个视窗向上 移动该窗口的一半高度。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)。
- 正向滚动一整屏的命令是 `CTRL-F` (减去两行)。反向的命令是 `CTRL-B`。`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窗中用作上 下文，可以使 用 `'scrolloff'` 选项。

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，输入 `"/` 时，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。

备注：字符 `.[*]^%&?~$` 有特殊含义。如果你要查找它们，需要在前面加上一个 `"`。

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`：

然后输入 `"n"` 数次。你会移动到其后每一个 `#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计数前缀。这样，`"3n"` 表示移动到第三个匹配点。要注意，`"/` 不支持计数前缀。

- 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计数: `"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
```

这样做, 下一次查找时你又需要切换回来。如果你只是想去掉高亮显示, 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。用这个功能可以检查是否会被找到正确的匹配，这时输入 `<Enter>` 就可以真正地跳到那个地方。否则，继续输入更多的字符可以修改要查找的字符串。

```
:set nowrapscan
```

这个设置使得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到文件头。

- 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，像下面这样：`: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并关闭文件：`ZZ`

现在如果你重新启动 Vim，`'hlsearch'` 选项就已经被设置了。

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。我们只介绍一些基本的正则表达式。

- 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
```

用 `"/the$"` 则匹配如下位置：

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

而使用 `"/^the` 则匹配:

the solder holding one of the chips melted and the

你还可以试着用这个模式: `"/^the$`; 它只会匹配仅包括 `"the"` 的行。并且不包括空格。例如包括 `"the "` 的行是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"` 匹配一个字符串, 它的第一个字符是 `c`, 第二个字符是任意字符, 而第三个字符是 `m`。例如:

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

- 匹配特殊字符

如果你确实想匹配点字符, 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."` 模式去查找, 会匹配这些地方:

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

但如果你查找 `"ter\"."`, 只会匹配第二个位置。

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:

```
``
```

`'` 是反引号, 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 `"'"` 命令本身是个跳转,

它记住了自己跳转前的位置。

一般, 每次你执行一个会将光标移动到本行之外的命令, 该移动即被称为一个 "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", 即使你在前面加上计数前缀使之移动到很远的地方也不例外。

`""` 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: 在很多常见的键盘布局上, `I` 在键盘上紧靠着 `O`)。考虑如下命令序列:

做小改动

1. [改变文本](#)
2. [重复一个修改](#)
3. [移动文本](#)
4. [拷贝文本](#)
5. [替换模式](#)
6. [返回目录](#)

改变文本

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" 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
```

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, 清楚但错误的回答。"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"。它实际上像 "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了这种方式, 这个不一致之处就留在 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缩进 (前导空格)。

"d\$" 删除到行尾;"c\$" 则修改到行尾。这相当于先用 "d\$" 删除一行再用 "a" 启动插入模式, 以便加入新的文字。

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" 命令不需要使用退出插入状态:

```
there is somerhing grong here
```

rT

rt

rw

```
There is something wrong here
```

通过计数前缀, "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"。这会删除一个字符并插入一个换行符。在这里使用计数前缀会删除多个字符但只插入一个换行符: "4r" 用一个换行符替换四个字符。

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 `<` 标记。你把光标移到第一个 `<` 上, 然后用 `"df>"` 命令删除。然后你就可以移到 `<` 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 `<` 标记, 移动到下一个 `<` 的位置, 再执行 `"."` 命令即可。

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

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, 放置)。

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。

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。

`"P"` 命令像 `"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"` 命令接受计数前缀, 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, 也就是在 `"h"` 的后面了。

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，计数前缀也是有效的。要拷贝两个单词，就可以用 **"y2w"**。

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。

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。

在下面的例子中，你在 **"text"** 的第一个 **"t"** 上启动替换模式：

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 在插入模式和替换模式间切换。

但当你使用 (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 "撤销" 命令一样。

使用语法高亮

1. [功能激活](#)
2. [颜色显示不出来或者显示出错误的颜色怎么办？](#)
3. [使用不同的颜色](#)
4. [返回目录](#)

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，并调用合适的语法高亮。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高亮，把 **"syntax enable"** 命令加入到 **vimrc** 文件中。如果你想语法高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" 放入你的 gvimrc 文件。

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 TERM
```

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
```

要知道哪些类型是有效的，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，这种方式使得该文件每次被编辑时都被高亮。例如，下面这行可用于 Makefile (把它放在接近文件首和文件末的地方)

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

你可能知道怎么检测自己的文件类型，通常的方法是检查文件的扩展名 (就是点后面的内容)。new-filetype 说明如何告知 Vim 进行那种文件类型的检查。

- 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。如果觉得不好，你可以自己写一个，参见 `mysyntaxfile`。

- 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。如果要在之后设置背景，可以再调用一下 `":syntax reset"` 使得 Vim 重新进行缺省颜色的设置。

- 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。

使用不同的颜色

如果你不喜欢默认的颜色方案，你可以选另一个色彩方案。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" 是色彩方案的名称。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的色彩方案，可以把 `":colorscheme"` 命令加到你的 `vimrc` 文件中。

你可以自己编写色彩方案，方法如下：

1. 选择一种接近你理想的色彩方案。把这个文件拷贝到你自己的 Vim 目录中。在 Unix 上，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. 编辑这个色彩方案，常用的有下面的这些条目：

指令	颜色
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 `"bold,underline"`。详细信息请参考 `:highlight` 命令。

3. 告诉 Vim 总使用你这个色彩方案。把如下语句加入你的 `vimrc` 中：

```
colorscheme mine
#如果你要测试一下常用的色彩组合，用如下命令：
:runtime syntax/colortest.vim
#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。
```

分割窗口

1. [分割窗口](#)
2. [用另一个文件分割窗口](#)
3. [窗口大小](#)
4. [垂直分割](#)
5. [移动窗口](#)
6. [对所有窗口执行命令](#)
7. [返回目录](#)

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:

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

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，像 `":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。

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 one.c */           |
|~                           |
|~                           |
one.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。

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。

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。

`'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。

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)。

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。

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，像这样：

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

还有三个相似的命令 (估计你已经猜出来了)：

```
CTRL-W H
CTRL-W J
CTRL-W L
```

把当前窗口移到最左边 把当前窗口移到最下边 把当前窗口移到最右边

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 74 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!
```

做大修改

1. [替换](#)
2. [读、写部分文件内容](#)
3. [排版文本](#)
4. [改变大小写](#)
5. [返回目录](#)

替换

`":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。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)?
```

`":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+
```

读、写部分文件内容

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, 你将被提示错误。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。

排版文本

在你输入纯文本时，自动换行自然会比较吸引的功能。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。先移到你要排版的行，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。

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

小窍门

1. [单词替换](#)
2. [排序](#)
3. [反转行顺序](#)
4. [单词统计](#)
5. [删除多余空格](#)
6. [返回目录](#)

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。如果你有一个像 "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"。

指令	指令解释
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

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 "OBJJS" 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：

```
OBJJS = \  
  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。

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
```

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。

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)。

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。

95后菜鸟码农，努力精进中，欢迎大家扫码加我好友共同学习成长，现在还可免费领取 10T的各类学习资料，同时可参与每日 免费包邮抽奖活动、现金红包 等，真诚无套路！

