Vim 中文用户手册

http://github.com/yianwillis/vimcdoc

March 29, 2025

usr_toc.txt 适用于 Vim 9.1 版本。 最近更新：2025年3月

VIM 用户手册 - by Bram Moolenaar

目 录 user-manual usr

总览

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（较老版本）的英文 HTML 版本和 PDF 版本可以从以下这个地址得到:

本手册的中文 HTML 版本和 PDF 版本可以从以下这个地址得到:
(HTML)
http://github.com/yianwillis/vimcdoc/releases (PDF)

初步知识

从头至尾阅读这些文档可以学习基本的命令。

usr_01.txt 关于本手册
- 01.1 手册的两个部分
- 01.2 安装了 Vim 之后
- 01.3 教程使用说明
- 01.4 版权声明

usr_02.txt Vim 初步
- 02.1 第一次运行 Vim
- 02.2 插入文本
- 02.3 移动光标
- 02.4 删除字符
- 02.5 撤销与重做
- 02.6 其它编辑命令
- 02.7 退出
- 02.8 寻求帮助

usr_03.txt 移动
- 03.1 词移动
- 03.2 移动到行首或行尾
- 03.3 移动到指定字符
- 03.4 括号匹配
- 03.5 移动到指定的行
- 03.6 确定当前位置
- 03.7 滚屏
- 03.8 简单查找
- 03.9 简单的查找模式
- 03.10 使用标记

usr_04.txt 做小改动
- 04.1 操作符与动作
- 04.2 改变文本
- 04.3 重复一个修改
- 04.4 可视模式
- 04.5 移动文本
- 04.6 拷贝文本

高效的编辑

可以独立阅读的主题。

调节 Vim

告诉 Vim 如何工作。

编写 Vim 脚本

让 Vim 工作

在你使用 Vim 之前。

---

 vim:tw=78:ts=8:noet:ft=help:norl:

usr_01.txt　　适用于 Vim 9.1 版本。　　最近更新：2025年3月

VIM 用户手册 - by Bram Moolenaar
译者：Nek_in

关于本手册

本章介绍 vim 的手册本身。读者可以通过本章来了解本手册是如何解释 Vim 命令的。

下一章：　usr_02.txt　　Vim 初步
目录：　usr_toc.txt

01.1　手册的两个部分

Vim 的手册分成两个部分：

1. 用户手册
   面向任务的使用说明书，由简入繁，能像书一样从头读到尾。

2. 参考手册
   详细描述 Vim 的每一个命令的详细资料。

本手册使用的符号请参见：　notation

跳 转

文本包含两部分内容之间的超级链接，允许你快速地在一个编辑任务的描述和它所涉及的命令和选项的详细说明之间跳转。可以使用如下两个命令实现这个功能：

    按 CTRL-] 跳转到当前光标下的单词的相关主题
    按 CTRL-O 回跳（重复这个操作可以回跳多次）

大部分链接放在两根竖线之间，如：　bars 。竖线本身可能被隐藏或不可见；见下。另外，'number' 形式的选项名，用双引号括住的命令 ":write" 或者其它任何单词都可以用作一个超级链接。不妨做如下试验：把光标移动到 CTRL-] 上，并按下 CTRL-]。

其它主题可以用 ":help" 找到；请参见　help.txt 。

竖线和星号在使用　conceal　功能时通常已被隐藏，它们也同时使用　hl-Ignore ，使文本色和背景色相同。要让它们重新可见：

```
:set conceallevel=0
:hi link HelpBar Normal
:hi link HelpStar Normal
```

01.2 安装了 Vim 之后

本手册大部分内容都假定 Vim 已经被正常安装了。如果还没有，或者运行不正常（例如，找不到文件或 GUI 模式下菜单没有显示出来等）。请先阅读关于安装的章节：
usr_90.txt 。

not-compatible

本手册还经常假定 Vim 运行在 Vi 兼容选项关闭的模式下。对大部分命令而言，这无关紧要，但有时却是很重要的。例如，多级撤销（undo）功能就是这样。有一个简单的方法可以保证你的 Vim 设置是正常的 - 拷贝 Vim 自带的 vimrc 范例。在 Vim 里进行操作，你就无需知道这个文件在什么地方。不同系统的操作方法分别如下：

对于 Unix:
```
:!cp -i $VIMRUNTIME/vimrc_example.vim ~/.vimrc
```
对于 MS-Windows:
```
:!copy $VIMRUNTIME/vimrc_example.vim $VIM/_vimrc
```
对于 Amiga:
```
:!copy $VIMRUNTIME/vimrc_example.vim $VIM/.vimrc
```

如果这个文件已经存在，你最好备份一下。

现在启动 Vim，'compatible' 选项应该已经关闭。你可以用以下命令检查一下：

```
:set compatible?
```

如果 Vim 报告 "nocompatible"，则一切正常；如果返回 "compatible" 就有问题。这时你需要检查一下为什么会出现这个问题。一种可能是 Vim 找不到你的配置文件。用如下命令检查一下：

```
:scriptnames
```

如果你的文件不在列表中，检查一下该文件的位置和文件名。如果它在列表中，那么一定是还有某个地方把 'compatible' 选项设回来了。

参考 vimrc 和 compatible-default 可以获得更多信息。

备注:
本手册是关于普通形态的 Vim 的，Vim 还有一种形态叫 "evim"（easy vim），那也是 Vim，不过被设置成 "点击并输入" 风格，就像 Notepad 一样。它总是处于 "插入" 模式，感觉完全不同于通常形态下的 Vim。由于它比较简陋，将不在本手册中描述。详细信息请参考 evim-keys 。

01.3 教程使用说明 tutor vimtutor

除了阅读文字（烦！），你还可以用 vimtutor（Vim 教程）学习基本的 Vim 命令，这是一个 30 分钟 2 个章节的教程，它能教会你大部分基本的 Vim 功能。

在 Unix 中，如果 Vim 安装正常，你可以从命令行上运行以下命令：

```
vimtutor
```

在 MS-Windows 中，你可以在 "Program/Vim 9.1" 菜单中找到这个命令，或者在安装目录（在 Vim 里运行 `:echo $VIMRUNTIME` 可以改到这个目录）中运行 vimtutor.bat 程序。

这个命令会建立一份教程文件第 1 章的拷贝，你可以任意修改它而不用担心会损坏原始的文件。要继续第 2 章的拷贝，可用以下命令：

```
vimtutor -c 2
```

这个教材有各种语言的版本。要检查你需要的版本是否可用，使用相应语言的双字母缩写。例如，对于法语版本：

```
vimtutor fr
```

在 Unix 上，如果你更喜欢 Vim 的 GUI 版本，可用 "gvimtutor" 或 "vimtutor -g" 来代替 "vimtutor"。

对于 OpenVMS 而言，如果已经正确安装了 Vim，VMS 提示行上可以启动 vimtutor：

```
@VIM:vimtutor
```

也可以加入可选的双字母语言代码，同上。

在非 Unix 系统中，你还要做些工作：

1. 拷贝教程文件。你可以用 Vim 来完成这个工作 (Vim 知道文件的位置)

```
vim --clean -c 'e $VIMRUNTIME/tutor/tutor1' -c 'w! TUTORCOPY' -c 'q'
```

这个命令在当前路径下建立一个 "TUTORCOPY" 的文件。要使用其它语言的版本，在文件名后加上双字母的语言缩写作为扩展名。对于法语：

```
vim --clean -c 'e $VIMRUNTIME/tutor/tutor1.fr' -c 'w! TUTORCOPY' -c 'q'
```

2. 用 Vim 编辑这个被拷贝的文件

```
vim --clean -c "set nocp" TUTORCOPY
```

--clean 参数用于保证 Vim 使用良好的缺省值启动。

3. 学习完成后删除临时拷贝文件

```
del TUTORCOPY
```

01.4 版权声明 manual-copyright

Vim 用户手册和参考手册的版权 (1988) 归 Bram Moolenaar 所有。这份材料只可以在符合开放出版物版权协议 (Open Publication License) 1.0 或以后版本的条件下发布。这份协议的最新版本可以在如下地址上找到：

https://opencontent.org/openpub/

对此文档作出贡献的人员必须同意这份声明。

frombook

这份文档的一部分来自 Steve Oualline 的《Vi IMproved - Vim》(New Riders出版，ISBN: 0735710015)。开放出版物版本协议同样适用于这本书。被选入的部分经过剪裁 (例如去掉插图，根据 Vim 6.0 和之后的新特性进行修改，并修正一些错误)。某些部分

省略了 frombook 标记，这并不表示这些部分不是来自这本书。

非常感谢 Steve Oualline 和 New Riders 编写并出版了这本书并在 OPL 协议下发行它！它对写这份手册起到了非常大的作用。它不只是提供了文字描述，还确定了这份手册的基调和风格。

如果你通过销售这份手册赚了钱，我们强烈建议你把部分利润捐给乌干达的艾滋病受害者。(参见 iccf)

(译者注：这里是对原文协议的翻译，中文翻译的版权，请参阅 vimcdoc 的 LICENSE 文件)

下一章： usr_02.txt Vim 初步

VIM 用户手册 - by Bram Moolenaar
译者：Nek_in

Vim 初 步

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，启动一个命令行窗口，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：

```
+----------------------------------------+
|#                                       |
|~                                       |
|~                                       |
|~                                       |
|~                                       |
|"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，编辑器就使用 xterm 窗口。如果你用的是 MS-Windows 的命令行提示窗口，编辑器就使用这个窗口。
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。

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。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，按回车开始一个新行。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。

摆 脱 麻 烦

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。而这些键位在你的右手本位。也就是说：这种键位的设定使你可以用最快的速度执行移动操作（特别是当你用十指输入的时候）。

> 备注：
> 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：

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

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
```

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
```

备注：
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
                        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。更多信

息可见 32.2 一节。

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，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:

```
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。
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。

02.7 退出

使用 "ZZ" 命令可以退出。这个命令保存文件并退出。

> 备注:
> 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。

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。

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!

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。

帮助的另一个优点是对应于你特定的 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
```

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" 将命令显示一个总览的帮助窗口。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，有这样的记号：*help.txt*。"*" 之间的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'
```

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，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
```

小结： **help-summary**

1) 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
```

2) 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
```
之后，你可以看看用户指南主题 03.9 和 usr_27.txt 相关的介绍。

3) 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，然后用正则表达式搜索，如 textwidth。
若干选项有自己的命名空间，例如：
```
:help cpo-<letter>
```
可查找 'cpoptions' 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 'guioptions' 的标志位：
```
:help go-<letter>
```

4) 普通模式命令没有前缀。如要转到 "gt" 命令的帮助页面：
```
:help gt
```

5) 插入模式命令以 i_ 开始。如关于删除单词的帮助：
```
:help i_CTRL-W
```

6) 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
```

7) 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_%
```

8) Ex-命令总是以 ":" 开始，如要转到 ":s" 命令的帮助：

```
    :help :s
```

9) 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
```

10) 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
```

带你到插入模式下的 CTRL-X 命令的家族，可用于自动补全不同的事物。注意，一些特定的键总是以相同的方式书写，如 Control 总写作 CTRL。
普通模式命令没有前缀，相关主题可用 :h CTRL-<Letter> 找到，如

```
    :help CTRL-W
```

与之作对比

```
    :help c_CTRL-R
```

描述 CTRL-R 在命令行上输入命令时的行为，而

```
    :help v_CTRL-A
```

讲述可视模式下数值的增量，而

```
    :help g_CTRL-A
```

讲述 "g<C-A>" 命令 (例如，你要先按 "g" 再按 <Ctrl-A>)。
这里 "g" 代表普通命令 "g"，它总期待按下第二个键后才能做事，"z" 开始的命令也类似。

11) 正则表达式项目总是以 / 开始。如要得到 Vim 正则表达式的 "\+" 量词的帮助:

```
    :help /\+
```

如果你需要知道所有关于正则表达式的情况，从这里开始:

```
    :help pattern.txt
```

12) 寄存器总是以 "quote" 开始。如要了解特殊的 ":" 寄存器:

```
    :help quote:
```

13) 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-!
```

会带你到描述 VimScript 中 "!" (取非) 操作符的主题。
同样重要

```
    :help function-list
```

提供所有可用函数的简短描述。Vim 脚本函数的帮助主题总包含 "()"，如:

```
    :help append()
```

讲述 Vim 脚本的 append 函数，而不是如何在当前缓冲区内附加文本。

14) 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
```

说明缓冲区局部的映射，或

```
    :help map-bar
```

说明如何在映射中处理 '|'。

15) :h command-topic 讲述命令的定义，所以用

```
    :help command-bar
```

可以了解关于自定义命令的 '!' 参数。

16) 窗口管理命令总是以 CTRL-W 开始，用 :h CTRL-W_letter 可以找到相应的帮助。如

:help CTRL-W_p
说明如何移动到上次访问过的窗口。也可访问
:help windows.txt
如果你在寻找窗口处理的命令，请仔细阅读。

17) 用 :helpgrep 在所有帮助页面中进行搜索（包括已安装的插件）。用法见 :helpgrep 。要搜索某主题：
:helpgrep topic
带你到首个匹配处。要到下一个：
:cnext
快速修复窗口包含所有的匹配，可以这样打开：
:copen
先移到你要的匹配，按 Enter 跳转到该帮助。

18) 用户手册。以初学者友好的方式描述帮助主题。 usr_toc.txt 可以找到目录（你可能猜到了），就从这里开始吧：
:help usr_toc.txt
粗略看看内容，找找感兴趣的主题。如 "二合字母" 和 "插入特殊字符" 项目出现在第 24 章，所以要转到该帮助页面：
:help usr_24.txt
另外要访问帮助中的特定章节，可以这样直接访问章节号：
:help 10.1
会转到 usr_10.txt 的第 10.1 章，那里讲述记录宏。

19) 高亮组。总是以 hl-groupname 开始。如
:help hl-WarningMsg
讲述 WarningMsg 高亮组。

20) 语法高亮使用命名空间 :syn-topic。如
:help :syn-conceal
讲述 ":syn" 命令的 conceal 参数。

21) 快速修复命令通常以 :c 开始，而位置列表命令通常以 :l 开始

22) 自动命令事件可用其名字查找：
:help BufWinLeave
要列出所有可能的事件：
:help autocommand-events

23) 命令行开关总是以 "-" 开始。如关于 Vim 的 -f 命令开关的帮助，可用：
:help -f

24) 可选的特性总是以 "+" 开始。如了解 conceal 特性，可用：
:help +conceal

25) 关于系统包含的文件类型的特定功能的文档，通常所在位置的形式是 ft-<文件类型>-<功能>。如
:help ft-c-syntax
讲述 C 语法文件和它提供的选项。有时，全能补全有自己的段落
:help ft-php-omni
文件类型插件也是如此，如
:help ft-tex-plugin
可以参考。

26) 错误和 警告 代码可在帮助中直接查找。这样
        :help E297
    准确地带你到交换错误信息的描述，而
        :help W10
    讲述 警告 "Changing a readonly file"。
    不过，有时这些错误代码没有相应描述，而是和通常产生该错误的 Vim 命令列在一起。如:
        :help E128
    带你到 :function 命令

27) Vim 发布的包的文本都有 package-<name> 形式。所以
        :help package-comment

    会带你到内含的注释插件的帮助小节，了解如何打开它。

下一章: usr_03.txt 移动

VIM 用户手册 - by Bram Moolenaar

译者：Nek_in

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。

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。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（从 "x" 标记的位置开始）：

```
This is a line with example text
  x-->-->->----------------->
   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
<----<--<-<---------<--x
   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
   <----<----x---->------------>
   2ge   ge     e       2e
```

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 认为是单词组成部分的字符，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&
```

还可以用空白字符分隔的 "字串"（大写的 WORD）移动。这不是我们通常意义的 "单词"。这就是为什么使用大写形式的 WORD 的原因。按字串移动的命令也全都是大写的，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，你可以在段落内快速前后移动。

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（"." 指示一个空格）：

```
            ^
       <-----------x
.....This is a line with example text
<----------------x   x-------------->
        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。

"0" 命令不能加计数前缀，因为 "0" 本身就是个数字。而且，出人意料地是，"^" 命令使用计数前缀也没有任何效果。

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" 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 "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" (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。

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: 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)
---+---------------->
                    %
```

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 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    |  G
               text text text text    |
               last line of a file    V
```

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。

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 (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)。

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. 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。

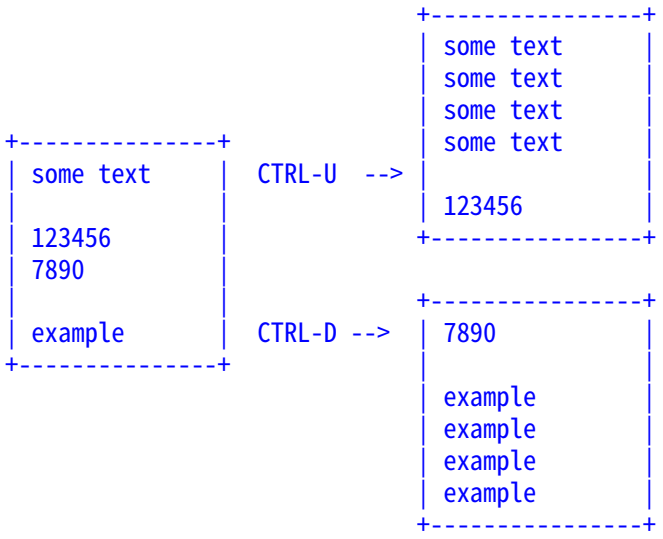
3. 置位 'ruler' 选项。这会在 Vim 窗口的右下角显示当前光标的位置：

```
: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。

03.7 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。

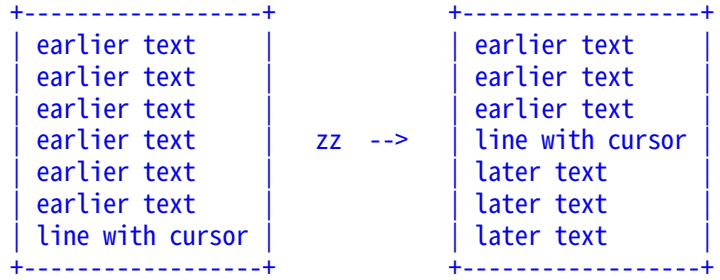


每次滚一行的命令是 CTRL-E（上滚）和 CTRL-Y（下滚）。可以把 CTRL-E 想象为是多给你一行（one line Extra）。

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。

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，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 "\"。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。"/" 也可用计数前缀：“4/the" 转到 "the" 的第四个匹配。

"?" 命令功能与 "/" 的功能类似，但进行反方向查找：

```
?word
```

"N" 命令在反方向重复前一次查找。因此，在 "/" 命令后执行 "N" 命令是反向查找，在 "?" 命令后执行 "N" 命令是正向查找。

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。

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，请参考第 27 章 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.
         xxx            xxx      xxx
```

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.
             xxxx                          xxxx
```

但如果你查找 "ter\."，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：

```
``
```

` 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：

```
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
```

备注：
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。最后一个你使用的标记会用 ">" 符号标记出来。

有 名 字 的 标 记 **bookmark**

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，后者同时移到标记标记的列上。

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。假设你在处理文件末的时候需要查看文件首的一些内容。

先移动到文件首，设置一个标记 s (start，开始):

```
ms
```

然后移动到你需要处理的地方，再设置一个标记 e (end，结束):

```
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件尾部的地方。

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:

```
'       跳转前的光标位置
"       最后编辑的光标位置
[       最后修改的开始位置
]       最后修改的结束位置
```

下一章: usr_04.txt 做小改动

usr_04.txt　　适用于 Vim 9.1 版本。　　最近更新：2020年1月

VIM 用户手册 - by Bram Moolenaar
译者：Nek_in

做小改动

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计数前缀，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" 命令作为动作来移动到单词结尾，Vim 这时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于你使用的移动命令。在参考手册中，当不包括这个字符时，称为 "非包含的" (exclusive)，而包括这个字符时，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
```

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。

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)
```

在什么地方加入计数前缀

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计数前缀，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<Enter>"。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<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" (撤销), CTRL-R (重做) 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"
.               重复修改到 "five" 的操作
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，要从一个单词的中间删除到下一个单词的中间:

```
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" 前清楚得看到将要被删除的是哪些文本。

如果任何时候你改了主意，只用按一下 <Esc> 就能退出可视模式。

按 行 选 择

如果你想对整行做操作，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    | |
选中的行   >>   | text text text         | | Vjj
           >>   | text more              | V
                | more text more         |
                +------------------------+
```

列 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 启动列块可视模式。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。

当使用列块可视模式的时候，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。

备注: "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，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。

```
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。

```
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，计数前缀也是有效的。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 菜单项目。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 菜单项目粘贴进来，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 (或者如果在行尾的话，单词前的空格字符)。

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-文本对象方法。

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" (内含句子)。还有一个文本对象是 "as" ("A Sentence")，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，而且继续留在可视模式，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 "撤销" 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。现在你已经知道它们是怎么工作了，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	删除到文件末
dgg	删除到文件首

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。

还有一些常用的进行修改的命令，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。

下一章： usr_05.txt 设置选项

usr_05.txt　　适用于 Vim 9.1 版本。　　最近更新：2025年3月

VIM 用户手册 - by Bram Moolenaar
译者：Nek_in、tocer

选项设置

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　使用语法高亮
前一章：　usr_04.txt　做小改动
目录：　usr_toc.txt

05.1　vimrc 文件　　vimrc-intro

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 会在启动的时候执行这个文件里的命令。

如果你已经有 vimrc 文件（例如，系统管理员已经为你配置好了），可以这样编辑它：

```
:edit $MYVIMRC
```

如果你还没有 vimrc 文件，请参考 vimrc 一节看看你应该在什么地方创建该文件。另外 ":version" 命令能告诉你 vim 要查找的 "用户 vimrc 文件" 的名字。

对于 Unix 和 Macintosh 系统，总是使用而且也推荐使用如下文件：

```
~/.vimrc
```

对于 MS-Windows，可以使用下面其中一个文件：

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

对于初学 vimrc 文件的朋友，建议在开头放上这行：

```
source $VIMRUNTIME/defaults.vim
```

这为 Vim 新用户进行了初始化（相对于传统 Vi 用户而言）。详见 defaults.vim 。

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。想了解更多关于 Vim 脚本的知识，请参见 usr_41.txt 。

05.2 vimrc 示例解释 vimrc_example.vim

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。

```
" 获取多数用户想要的缺省值。
source $VIMRUNTIME/defaults.vim
```

这会载入 $VIMRUNTIME 目录里的 "defaults.vim" 文件。这会按照多数用户的喜好设置 Vim。如果你是少数不喜欢的，注释掉这行。其中的命令解释见下：
defaults.vim-explained

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

如果可用，也设置 'undofile' 选项。会在一个文件中保存多层撤销信息。效果是，当你改动了文件，退出 Vim，然后再次编辑文件时你可以撤销之前做过的改动。这是很强大很有用的功能，代价是要保存一个文件。详情可见 undo-persistence 。

"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 。

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

它打开 'hlsearch' 选项，告诉 Vim 高亮上次查找模式匹配的地方。

```
augroup vimrcEx
  au!
  autocmd FileType text setlocal textwidth=78
augroup END
```

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。

"augroup vimrcEx" 和 "augroup END" 的封装使 "au!" 命令可以删除自动命令。见 :augroup 。

```
if has('syntax') && has('eval')
  packadd! matchit
endif
```

如果所需的特性可用，这会载入 "matchit" 插件。这使得 % 命令更强大。 matchit-install 有解释。

05.3 defaults.vim 文件解释 defaults.vim-explained

如果用户没有 vimrc 文件，载入 defaults.vim 文件。如要你创建自己新的 vimrc 文件，在顶部附近加入此行，以继续使用：

```
source $VIMRUNTIME/defaults.vim
```

或者使用上面解释过的 vimrc_example.vim 文件。

下面解释 defaults.vim 做的事情。

```
if exists('skip_defaults_vim')
  finish
endif
```

defaults.vim 的载入可以用这个命令关闭：

```
let skip_defaults_vim = 1
```

必须在系统 vimrc 文件完成。见 system-vimrc 。如果有你自己的用户 vimrc，不必如此，因为不会自动载入 defaults.vim。

```
set nocompatible
```

就像第一章所述，本手册解释在改进的方式下工作的 Vim，因此与 Vi 不完全兼容。要关闭 'compatible' 选项，'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在哪里允许 <BS> 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，换行符和插入模式开始处之前的字符。见 'backspace'。

```
set history=200
```

这个命令保存 200 个命令和 200 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。见 'history'。

```
set ruler
```

总在 Vim 窗口的右下角显示当前光标位置。见 'ruler'。

```
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 wildmenu
```

在状态行上显示补全匹配。这是按了 <Tab> 后有多于一个匹配的情况。见 'wildmenu'。

```
set ttimeout
set ttimeoutlen=100
```

这使得按 Esc 的生效更快速。通常 Vim 要等待一秒来看看 Esc 是否是转义序列的开始。如果你使用很慢的远程连接，增加此数值。见 'ttimeout'。

```
set display=truncate
```

如果末行被截短，显示 @@@ 而不是隐藏整行。见 'display'。

```
set incsearch
```

在输入部分查找模式时显示相应的匹配点。见 'incsearch'。

```
set nrformats-=octal
```

不把零开头的数值识别为八进制数。见 'nrformats'。

```
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。

```
inoremap <C-U> <C-G>u<C-U>
```

插入模式下的 CTRL-U 删除当前行所有输入的文本。使用 CTRL-G u 会先打断撤销，这样你可以在插入换行符后撤销 CTRL-U。用 ":iunmap <C-U>" 还原。

```
if has('mouse')
  set mouse=a
endif
```

如果可用，允许使用鼠标。见 'mouse'。

```
vnoremap _g y:exe "grep /" .. escape(@", '\\/') .. "/ *.c *.h"<CR>
```

这个映射在可视模式下拷贝已选择的文本并在 C 文件中查找它。你可以看到该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。

```
syntax on
```

打开文件的色彩高亮。见 syntax 。

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高亮和以下另两项。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'。

restore-cursor last-position-jump

```
augroup RestoreCursor
  autocmd!
  autocmd BufReadPost *
    \ let line = line("'\"")
    \ | if line >= 1 && line <= line("$") && &filetype !~# 'commit'
    \      && index(['xxd', 'gitrebase'], &filetype) == -1
    \ |   execute "normal! g`\""
    \ | endif
augroup END
```

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。这里跳过 commit 或 rebase 信息，因为每次的信息可能都不同，也不适用于 xxd(1) 过滤和编辑二进制文件，因为要对输入文件来回转换，所以可以这么说，这里有双重性。另见 using-xxd 。

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。

```
command DiffOrig vert new | set bt=nofile | r ++edit # | 0d_ | diffthis
          \ | wincmd p | diffthis
```

它加入 ":DiffOrig" 命令。在修改过的缓冲区中应用可以看到和原先载入的文件间的差异。见 diff 和 :DiffOrig 。

```
set nolangremap
```

防止 langmap 选项应用于映射生成的字符上。如果置位（缺省），会破坏插件（但后向兼容）。见 'langremap'。

05.4 简单键盘映射

映射可以使你把一系列 Vim 命令绑定为一个单独的键。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>
```

备注：
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。读这份手册时请注意这些区别！

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 {}，只需要移到单词的第一个字符上并按 F5。

在这个例子中，触发器是单独一个键；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，该命令将不再有效。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" 为在单词两端加圆括号，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 。

05.5 添加软件包　　add-package　matchit-install　package-matchit

软件包是一组可加入 Vim 的文件。有两种软件包：可选的和启动时自动载入的。

Vim 发布提供了一些软件包供你选用。例如 matchit 插件。此插件使 "%" 命令跳转到匹配的 HTML 标签、Vim 脚本中的 if/else/endif 等。很有用，但不后向兼容（这是为什么缺省没有打开）。

要开始用 matchit 插件，在 vimrc 文件中加入一行：

```
packadd! matchit
```

就这样！重启 Vim 后可以找到此插件的帮助：

```
:help matchit
```

这之所以能工作，是因为 :packadd 载入插件时，同时把软件包目录加入 'runtimepath'，这样就能找到帮助文件了。

网上有多个地方可以找到软件包。它们通常以归档或版本库形式出现。如果是归档，可以采用以下步骤：

1. 建立软件包目录：
   ```
   mkdir -p ~/.vim/pack/fancy
   ```
   "fancy" 可以是任何你喜欢的名字。要能描述你的软件包。
2. 在该目录里解压归档。假定归档的顶层目录是 "start"：
   ```
   cd ~/.vim/pack/fancy
   unzip /tmp/fancy.zip
   ```
   如果归档的目录结构不同，确保最终的路径像这样：
   ```
   ~/.vim/pack/fancy/start/fancytext/plugin/fancy.vim
   ```
   这里 "fancytext" 是软件包名，这可以是任何其它名字。

添加 editorconfig 包　　editorconfig-install　package-editorconfig

和 matchit 包类似，要在 Vim 启动时载入发布的 editorconfig 插件，在 vimrc 文件

中加入一行:
	packadd! editorconfig

重启 Vim 后，此插件已激活，相关阅读可见:
	:h editorconfig.txt

添加 comment 包 comment-install package-comment

用此命令载入插件:
	packadd comment

这样可绑定 gc 和类似的缺省键用于注释（在 Vim 社区时这是公认的映射)。

如果在 vimrc 文件里加入上述行，要重启 Vim 才能载入此包。一旦载入了包，可以这样了解它:
	:h comment.txt

添加 nohlsearch 包 nohlsearch-install package-nohlsearch

用此命令载入插件:
	packadd nohlsearch

'updatetime' 或进入 Insert 模式后自动执行 :nohlsearch 。
这里假定缺省的 updatetime，hlsearch 会在 4 秒的闲置时间后暂停/关闭。

要在载入后关闭此插件的效果:
	au! nohlsearch

关于软件包的详情可见: packages 。

05.6 添加插件 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。
{仅当 Vim 编译时加入 +eval 特性时才有效}

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 distributed-plugins

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 。

同样地，本地安装的插件和包（如果自带单独的帮助文件的话），也可在帮助的 local-additions 小节里找到类似的列表。

还可以参照 load-plugins 。

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？
- 有一些会总是被载入，你可以在 $VIMRUNTIME/plugin 目录中找到它们。
- 有一些与 Vim 一起发布，你可以在 $VIMRUNTIME/macros 目录或其子目录和 $VIM/vimfiles/pack/dist/opt/ 中找到。
- 从网上下载， 收集了很多。
- 在 Vim 的邮件列表中找： maillist 。
- 自己写一个，参见 write-plugin 。

某些插件被打包在 vimball 中，参见 vimball 。
某些插件可以自动更新，参见 getscript 。

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	$HOME/vimfiles/plugin 或 $VIM/vimfiles/plugin
Amiga	s:vimfiles/plugin
Macintosh	$VIM:vimfiles:plugin

以 Unix 系统为例（假设你还没有 plugin 目录）：

```
mkdir ~/.vim
mkdir ~/.vim/plugin
cp /tmp/yourplugin.vim ~/.vim/plugin
```

就是这样了！现在你可以用这个插件定义的命令了。

除了把这些插件直接放进 plugin/ 目录以外，还可以更好地组织一下，把它们放进 plugin 的单独的子目录中。例如，可以考虑把所有 Perl 插件放置在 "~/.vim/plugin/perl/*.vim"

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 兼容的文件系统上不能使用长文件名。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 上，"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。
注意 插件必须以 ".vim" 结尾。

请进一步阅读:

filetype-plugins	文件类型插件的文档和有关如何避免映射引起的问题。
load-plugins	全局插件的启动时间。
ftplugin-overrule	控制全局文件类型插件的选项。
write-plugin	如何写插件。
plugin-details	关于如何使用插件的信息或者当你的插件不工作的时候如何处理。
new-filetype	如何检测新文件类型。

05.7 添加帮助 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 "doit.vim" 插件为例。这个插件有一个文档: "doit.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。(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 /tmp/doit.vim ~/.vim/plugin
```

"cp" 命令是基于 Unix 的，MS-Windows 上你可以用 "copy"。

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 /tmp/doit.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 doit
```

从你刚才加上的帮助文件中获得 "doit" 的帮助了。在使用如下命令的时候，可以看到一个新的条目:

```
: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 。

05.8 选项窗口

如果要找一个选项，你可以在这里寻找帮助： 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。这些选项根据种类分组。将光标移到一个主题上然后按 <Enter> 就可以跳转到那里。再按一下 <Enter> 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。

这行的上方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。

对于那些值为数值或者字符串的选项，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。

05.9 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 (&) 把它恢复到默认值。例如：

```
:set iskeyword&
```

禁 止 回 绕 行

Vim 通常会对长行自动回绕，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。以下命令可以关闭行的回绕：

```
:set nowrap
```

当你移动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本，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，要在插入模式中使 <Left> 和 <Right> 也有效，可以：

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

还有一些可以用的标志位，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 ">---" ("-" 的个数不定) 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' 选项指定哪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" 是可显示的拉丁字符。
    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，逗号又自动被删除了。

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。

下一章: usr_06.txt 使用语法高亮

usr_06.txt　　适用于 Vim 9.1 版本。　　最近更新：2022年8月

VIM 用户手册 - by Bram Moolenaar
译者：Nek_in

使用语法高亮

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。

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 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 。

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" 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。

06.3 使用不同颜色 :syn-default-override

如果你不喜欢默认的色彩方案，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：

```
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。不过这些不是仅有的可用颜色。可以指定 #rrggbb 十六进制颜色，也可以为十六进制颜色在 v:colornames 里定义新颜色名，如：

```
let v:colornames['mine_red'] = '#aa0000'
```

如果你想编写别人都能用的色彩方案，注意 仅当颜色尚不存在时添加定义：

```
call extend(v:colornames, {'mine_red': '#aa0000'}, 'keep')
```

这使色彩方案的用户可在载入你的色彩方案之前，覆盖该颜色的准确定义，例如，在 .vimrc 文件里：

```
runtime colors/lists/css_colors.vim
let v:colornames['your_red'] = v:colornames['css_red']
colorscheme yourscheme
```

作为色彩方案作者，应该可以依赖用于 GUI 颜色的颜色名。它们定义在 colors/lists/default.vim 。每次运行 colorscheme 命令时都载入所有这些在 'runtimepath' 上找到的文件。vim 发布提供的官方列表应该已包含所有的 X11 颜色(之前在 rgb.txt 中定义)。

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。

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区的所有语法高亮。详见 :syntax-off 。

:syn-manual

如果你想只对特定的文件采用语法高亮，可以使用这个命令：

```
:syntax manual
```

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

```
:set syntax=ON
```

06.5 带颜色打印 syntax-printing

在 MS-Windows 版本上，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再执行打印命令，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，然后用浏览器打印。

如下命令把当前文件转换成 HTML 格式:

```
:TOhtml
```

如果不行:

```
: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。

06.6 深入阅读

usr_44.txt 自定义语法高亮

syntax		关于本话题的全部细节

下一章: usr_07.txt 编辑多个文件

usr_07.txt　　适用于 Vim 9.1 版本。　　最近更新：2020年3月

VIM 用户手册 - by Bram Moolenaar
译者：Nek_in

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。其输出应该像下面这样：

```
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 03.10 一节，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" 命令像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。

下一章： usr_08.txt 分割窗口

VIM 用户手册 - by Bram Moolenaar
译者：Nek_in

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：

```
+----------------------------------+
|/* 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。

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另一个更小。

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：

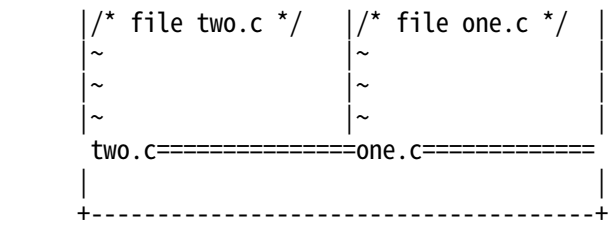
```
:vsplit
```

或者

```
:vsplit two.c
```

这个命令的结果如下：

```
+-------------------------------------+
```



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：

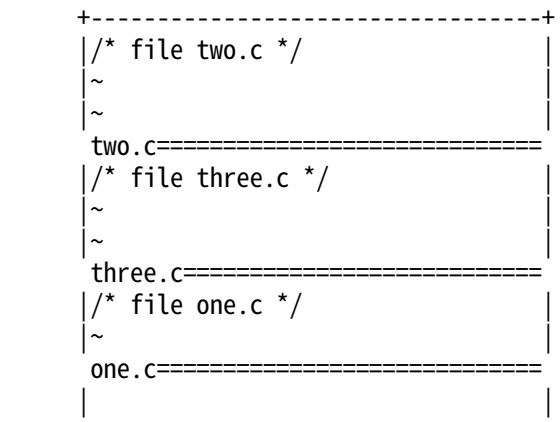
```
CTRL-W h        跳转到左边的窗口
CTRL-W j        跳转到下面的窗口
CTRL-W k        跳转到上面的窗口
CTRL-W l        跳转到右边的窗口

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，像这样：



```
+---------------------------------+
```

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：

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

当你在中间的窗口（three.c）中使用 CTRL-W K 后，结果会是：

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

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，这个命令是不能撤销的。

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                |
|~                             |
 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。

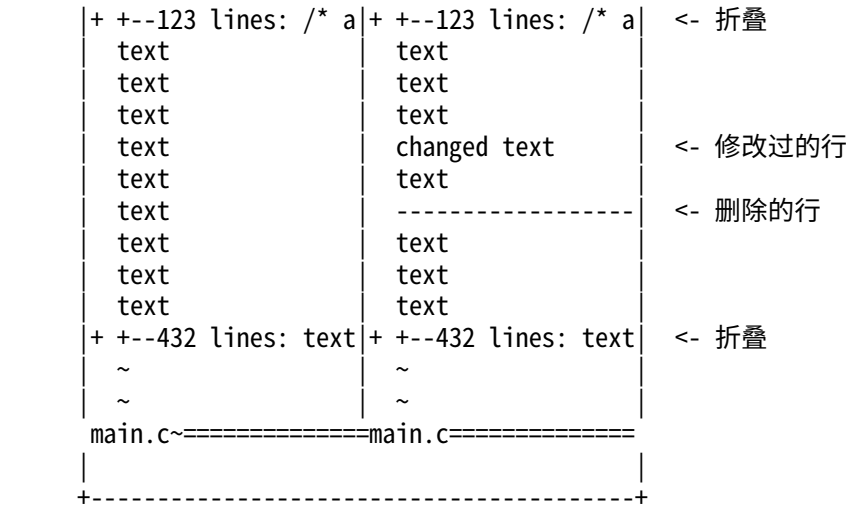
08.7 用 vimdiff 显示文件差异

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。

```
 VV                  VV
+--------------------------------------+
```



(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 。

下一章: usr_09.txt 使用 GUI 版本

usr_09.txt　　适用于 Vim 9.1 版本。　　最近更新：2019年12月

VIM 用户手册 - by Bram Moolenaar
译者：Nek_in

使用 GUI 版本

Vim 能在一般的终端中很好地工作。gVim 则提供了图形用户界面 (GUI)。它可以完成相同，甚至更多的功能。GUI 能提供菜单，工具条，滚动条和其它东西。本章介绍 GUI 这些额外的功能。

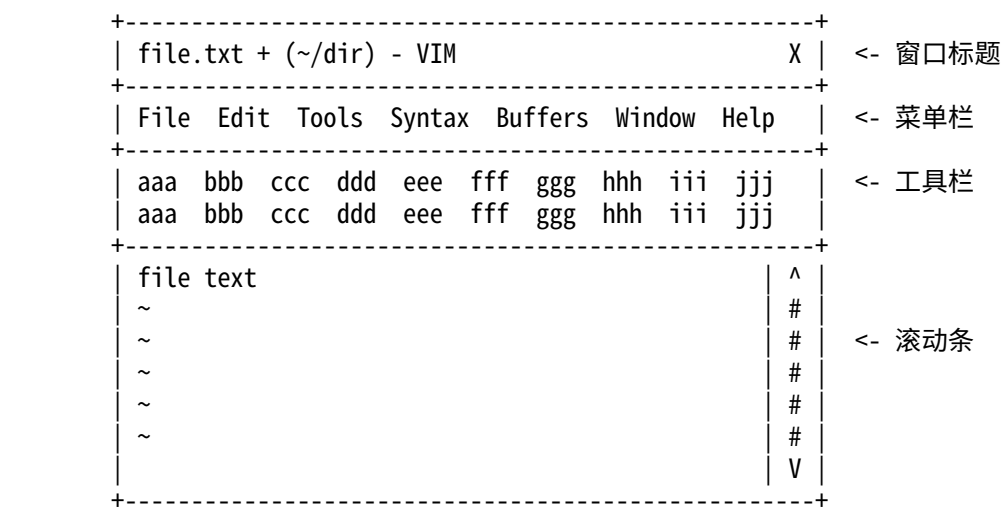
09.1　GUI 版本的组件

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，就是那个看起来像条虚线的。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像 X Windows 或者像微软 Windows 的鼠标。下面的命令使鼠标用起来像 X Windows：

```
:behave xterm
```

而如下命令使鼠标用起来像微软 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 gvim 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。选中的文字像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。

下一章： usr_10.txt 做大修改

VIM 用户手册 - by Bram Moolenaar
译者：Nek_in

做大修改

第四章我们已经介绍过做小修改的方法了。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像 "@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}                 像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}
```

这有点像 ":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。也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。

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。

注意：'!cmd!（不带文件范围的用法）和 "{range}!cmd" 有区别。前者会简单地执行外部命令，Vim 显示结果，而后者会通过过滤命令过滤 {range} 行，并替换范围内的文本为过滤命令的结果。详见 :! 和 :range! 。

如果命令不执行怎么办

启动一个外壳，发送一个命令并捕获它的输出，这需要 Vim 知道这个外壳程序是怎么工作的。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外壳。
        'shellcmdflag'  传给外壳的参数
        'shellquote'    外壳程序使用的引号（用于引用命令）
        'shellxquote'   用于命令和重定向文件名的引号
        'shelltype'     外壳程序的类型（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。因为总共只有两种外壳程序："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，有很多不同的外壳程序，所以你必须修改这些外壳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
```

下一章: usr_11.txt 从崩溃中恢复

VIM 用户手册 - by Bram Moolenaar
译者：Nek_in

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"）

备注 在下面情形下，Vim 知道交换文件没有用了，会自动删除它：
- 文件是合法的交换文件（魔术数正确）。
- 文件修改过的标志位未置位。
- 进程不在运行。

可以用 FileChangedShell 自动命令事件来编程处理这种情况。

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，你可以从对话框的六个选择中挑一个：

```
  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 。
要通过程序访问交换文件，见 swapinfo() 。

11.4 深入阅读

swap-file	解释交换文件在什么地方创建以及名字是什么。
:preserve	手工刷新交换文件
:swapname	查看当前文件的交换文件

'updatecount'	多少个键被敲下后执行一次交换文件刷新
'updatetime'	交换文件刷新后的超时时间
'swapsync'	交换文件刷新后是否执行磁盘同步
'directory'	列出用于保存交换文件的目录
'maxmem'	写入交换文件前的内存使用限制
'maxmemtot'	同上，当用于所有文件

下一章: usr_12.txt 小窍门

VIM 用户手册 - by Bram Moolenaar
译者：Nek_in

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。但是使用记录和回放命令会快很多。
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	*

匹配逗号空格 ", "   ,
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/^/move 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想像中的第 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。见 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 的外壳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。

下一章: usr_20.txt 快速键入命令行命令

VIM 用户手册 - by Bram Moolenaar
译者：Chimin Yen

快速键入命令行命令

Vim 具备若干基本功能，以简化键入命令的工作。你可以缩写，编辑和重复冒号命令，而补全功能几乎可以用于所有的场合。

20.1　命令行编辑

当你用冒号（:）命令或用 / 或 ? 搜索一个字符串时，Vim 就会把光标置于屏幕下方。你在那儿键入命令或者搜索模式。此处即称为命令行，也是用来输入搜索命令的地方。

最为显而易见的编辑命令的方法是按 <BS> 退格键。按下此键即删去光标前面的字符。如果要删去另一个更早键入的字符，得先用光标方向键把光标移到那儿。

例如，你键入了：

```
:s/col/pig/
```

在你按下回车键之前，你注意到 "col" 应为 "cow"。为了纠正这个错误，你按五次 <Left> 键。现在，光标正好在 "col" 后面。按 <BS> 键，然后键入正确的字符 "w"：

```
:s/cow/pig/
```

现在你可以立刻按 <Enter> 键了。在执行这个命令之前，你无须先把光标移到命令行的末尾。

命令行上移动光标时最常用的键：

<Left>	左移一个字符
<Right>	右移一个字符
<S-Left> 或 <C-Left>	左移一个单词
<S-Right> 或 <C-Right>	右移一个单词
CTRL-B 或 <Home>	命令行行首
CTRL-E 或 <End>	命令行行尾

备注：
<S-Left>（光标左移键和 Shift 键同时按下）和 <C-Left>（光标左移键和

Control 键同时按下）并非在所有键盘上都有效。其它 Shift 和 Control 组合键也是这种情况。

你也可以用鼠标来移动光标。

删 除

如前所述，<BS> 键删除光标前一个字符。删除光标前整个单词，则用 CTRL-W。

```
/the fine pig

            CTRL-W

/the fine
```

CTRL-U 删除命令行上全部文字，从而让你从头开始。

替 换

插入键 <Insert> 让你在插入字符和替换字符两种方式之间切换。先键入如下文字：

```
/the fine pig
```

再按两次 <S-Left>（或 <S-Left> 无效时按八次 <Left>），把光标移到 "fine" 起始处。现在，按插入键 <Insert>，切换到替换方式，并键入 "great"：

```
/the greatpig
```

哟，空格没了。现在，别用 <BS> 键，因为那会删除 "t"（这跟插入方式不同）。此时应该按插入键 <Insert>，从替换方式切换到插入方式，并键入空格：

```
/the great pig
```

取 消

你本想执行一个 : 或 / 命令，但却改变了主意。要清除命令行上你已经键入的文字却不执行该命令，按 CTRL-C 或 <Esc>。

> 备注：
> <Esc> 是普遍采用的 "退出" 键。不幸的是，在过去美好的 Vi 版本里，在命令行上按 <Esc> 却是执行命令！由于那会被认为是程序 bug，Vim 采用 <Esc> 来取消命令。但其 'cpoptions' 选项可以使 Vim 跟 Vi 兼容。而且，使用映射时（那可能是为 Vi 而写的），<Esc> 键也总和 Vi 兼容。由此看来，采用 CTRL-C 倒不失为一种永远奏效的方法。

如果你在命令行开始处，那么按 <BS> 将取消整个命令。这就像删除命令行赖以开始的 ":" 或 "/"。

20.2 命令行缩写

有些 ":" 命令确实很长。我们已经提及替代命令 ":substitute" 可以被缩写成 ":s"。这是一个基本机理，即所有 ":" 冒号命令都可以被缩写。

一个命令可以被缩写成多短呢？英文有 26 个字母，而 Vim 却有多得多的命令。例如，":set" 也以 ":s" 开头，但 ":s" 不是启动 ":set" 命令的。":set" 可以被缩写成 ":se"。

把一个命令缩写得更短即可能适用于两个命令，此时，该缩写仍然只能代表其中的一个。而选择哪个却没有任何逻辑，你不得不一个一个地记。最短的有效缩写形式可以在帮助文件里找到。例如：

```
:s[ubstitute]
```

它的意思是替代命令 ":substitute" 的最短缩写形式是 ":s"。随后的字符可以任选。因此 ":su" 和 ":sub" 都有效。

在用户手册里我们有时用命令的全称，有时简称意义明白就用简称。例如，":function" 可以缩写成 ":fu"。但既然大多数读者不明其所指，所以我们采用简称 ":fun"。(Vim 没有命令 ":funny"，否则简称 ":fun" 也会令人迷惑。)

在 Vim 脚本里你最好用命令全称。当你日后改编脚本时，全称读起来比较容易。除非那是一些常用命令如 ":w" (":write") 和 ":r" (":read")。

一个特别令人迷惑的缩写是 ":end"，它可以指 ":endif"，":endwhile" 或 ":endfunction"。所以，遇到这类命令时，最好永远采用全称。

选 项 简 称

在用户手册里，我们采用选项的全称。许多选项还有简称。跟 ":" 冒号命令不一样，有效的选项简称只有一个。例如，'autoindent' 的简称是 'ai'。因而下面两个命令完成同样的动作：

```
:set autoindent
:set ai
```

你可以从这儿找到完整的选项全称和简称的列表： option-list 。

20.3 命令行补全

命令行补全是那些仅仅因为它就值得从 Vi 转到 Vim 的功能之一。一旦你用上了这个功能，你就离不开它了。

假定你有个文件目录，其中存放了这些文件：

```
info.txt
intro.txt
bodyofthepaper.txt
```

你用这个命令来编辑最后那个文件：

```
:edit bodyofthepaper.txt
```

这很容易打错。一个快得多的方法是：

```
:edit b<Tab>
```

其结果是同样的命令。这是怎么回事？制表键 <Tab> 会补全光标前的词。在本例中就是 "b"。Vim 在目录中寻找并找到了唯一的一个以 "b" 开头的文件。那个文件想必是你寻找的，因此 Vim 为你补全了文件名。

现在键入：

```
:edit i<Tab>
```

Vim 会鸣起响铃，并给你这个结果：

```
:edit info.txt
```

响铃的意思是 Vim 找到了不止一个匹配。然后它使用了找到的第一个匹配（按字母顺序）。如果你再按一次 <Tab>，你得到：

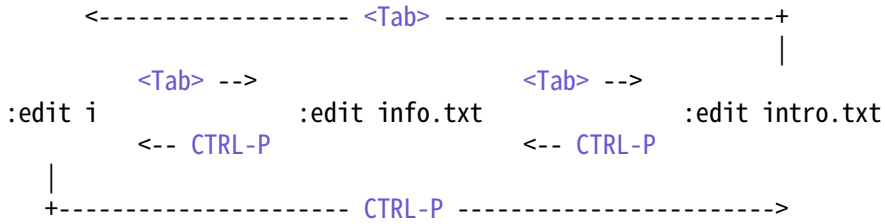
```
:edit intro.txt
```

这样，如果第一次 <Tab> 没给你你要找的文件，你就再按一次。如果还有匹配的文件，你将会看到它们每一个，每按一次，就看到一个。

如果你在最后一个匹配文件名上按 <Tab>，你将会再次看到你起初键入的那个命令：

```
:edit i
```

然后一切再从头开始。这样，Vim 就在匹配列表内周而复始地循环。使用 CTRL-P 依相反方向循环匹配列表：



上 下 文

当你键入 ":set i" 而不是 ":edit i"，并按 <Tab>，你得到的是：

```
:set icon
```

嗨，为什么你没得到 ":set info.txt"？这是由于 Vim 的补全功能是上下文相关的。Vim 寻找的那类词取决于关键词前面的命令。Vim 知道你不可能在命令 ":set" 后面用一个文件名，但却可以用一个选项名。

同样，如果你重复键入 <Tab>，Vim 就会在所有匹配间周而复始地循环。你最好开始时多键入几个字符，否则匹配的选项会很多：

```
:set isk<Tab>
```

结果是：

```
:set iskeyword
```

现在键入 "=" 并按 <Tab>:

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

在此，Vim 插入的是该选项原来的设定值。现在你可以编辑它了。
按 <Tab> 键补全的乃是 Vim 在命令行那个位置上所期待的。你不妨试试，看它究竟是如何补全的。在某些情形下你会得不到你想要的结果。那或者因为 Vim 不知道你要什么，或者因为 Vim 还没有为该情况实现补全功能。在那个场合，你的 <Tab> 就会被当作字符插入文本（显示为 ^I）。

匹配列表

当匹配有很多时，你可能希望看到一个总览。要看匹配总览，请按 CTRL-D。例如，你键入了以下命令以后按 CTRL-D:

```
:set is
```

结果是:

```
:set is
incsearch  isfname    isident    iskeyword  isprint
:set is
```

Vim 列出了匹配，然后又伴随着你早先键入的命令回到命令行。你现在可以检验匹配列表，找你想要的选项了。如果它不在列表中，你可以用 <BS> 修正那个词。如果匹配太多了，就在按 <Tab> 键补全命令之前再多打几个字符。
如果你仔细观察了，你就会注意到选项 "incsearch" 不是以 "is" 开头。在这个场合，"is" 指的就是 "incsearch" 的简称。(许多选项都有其简称和全称。) Vim 知道你可能要把选项的简称扩展成全称，够聪明的吧。

更多选项

命令 CTRL-L 把词补全为最长无歧义字符串。如果你键入 ":edit i" 而目录内存放着文件 "info.txt" 和 "info_backup.txt"，那么，你将得到 ":edit info"。

选项 'wildmode' 可用来改变补全时采用的方式。
选项 'wildmenu' 可用来取得菜单式的匹配列表。
利用选项 'suffixes' 来指定一些不太重要的文件，并让它们出现在文件列表末尾。
选项 'wildignore' 指定一些根本用不着表列的文件。

欲知更多此类选项详情，参见： cmdline-completion

20.4 命令行历史

我们曾经在第三章简单提到过历史记录功能。其基本用法就是你可以用 <Up> 键调用较早的命令行。而 <Down> 键则让你回到较晚的命令行。

历史记录功能实际上共有五种。我们在这里要提到的是用于 ":" 冒号命令以及用于 "/" 和 "?" 搜索命令的历史记录功能。"/" 和 "?" 命令共享同一历史记录。因为两者同为搜索命令。另外三种历史记录功能分别用于表达式、调试模式命令和 input() 函数的输入

行。 cmdline-history

假定你完成了一个 ":set" 命令，又键入了十个冒号命令，然后要再次执行那个 ":set" 命令。你本可以按一个 ":"，然后按十次 <Up>。更快捷的方法是:

```
:se<Up>
```

你现在就回到以前执行过的，以 "se" 开头的命令。没准儿那就是你想找的 ":set" 命令。至少你不至于按很多 <Up> 键 (除非你执行的都是 ":set" 命令)。

<Up> 键用你在命令行上已经键入的文本去跟历史记录里的命令作比较。只有匹配的命令才被显示出来。
    如果你没找到你要找的命令，用 <Down> 回到你键入的文本作修改。或者用 CTRL-U 从头来过。

显示历史记录里所有的命令:

```
:history
```

那是 ":" 冒号命令的历史记录。搜索历史记录用这个命令来显示:

```
:history /
```

如果你不愿用光标方向键，CTRL-P 作用就跟 <Up> 一样。而 CTRL-N 跟 <Down> 一样。CTRL-P 意指前一个 (previous)，CTRL-N 意指下一个 (next)。

20.5 命令行窗口

在命令行上键入文本跟插入模式下键入文本有所不同。许多修改文本的命令都不能用。对大多数命令而言，那问题不大，但有时你必须键入一个复杂的命令。那样的场合用命令行窗口就非常有用了。

用这个命令来打开命令行窗口:

```
q:
```

Vim 现在就在屏幕底部打开了一个 (很小的) 窗口。它存放着命令行历史记录，以及一行空行在末尾:

```
+-----------------------------------+
|other window                       |
|~                                  |
 file.txt============================
|:e c                               |
|:e config.h.in                     |
|:set path=.,/usr/include,,         |
|:set iskeyword=@,48-57,_,192-255   |
|:set is                            |
|:q                                 |
|:                                  |
 command-line========================
|                                   |
+-----------------------------------+
```

现在你处于普通模式下。你可以用 "hjkl" 键来移动光标。例如，用 "5k" 上移至
":e config.h.in" 那一行。键入 "$h" 移到 "in" 的 "i" 字符上，并键入 "cwout"。
现在你把这一行改成了这样：

```
:e config.h.out
```

现在按 <Enter> 执行这个命令。命令行窗口就关上了。
按 <Enter> 键执行光标下的那个命令。这跟 Vim 处于插入模式抑或普通模式无关。
　在命令行窗口内所作的修改不会被保存。它们不会导致历史记录被修改。唯一例外就是你执行的命令将被加到历史记录的末尾，跟所有执行过的命令一样。

在你要总览历史记录时，命令行窗口十分有用，查找类似命令，修改一点点，然后执行它。一个搜索命令可用来寻找某些东西。
　在前面那个例子中，搜索命令 "?config" 是可以用来寻找先前那个含有 "config" 的命令的。这有点儿特别，因为你是在命令行窗口内用命令行作搜索。在键入搜索命令时，你打不开另一个命令行窗口，命令行窗口是唯一的。

下一章： usr_21.txt 离开和回来

VIM 用户手册 - by Bram Moolenaar
译者：Chimin Yen

离开和回来

本章深入讨论如何让其它程序跟 Vim 混用。或者从 Vim 内部执行别的程序，或者暂时先离开 Vim 而于执行了那个程序后再回来。而且，本章将进一步介绍如何记住 Vim 的当前状态，并在以后把它还原回来。

下一章：　usr_22.txt　　寻找要编辑的文件
前一章：　usr_20.txt　　快速键入命令行
目录：　usr_toc.txt

21.1　挂起和继续

像多数 Unix 程序一样，Vim 可以按 CTRL-Z 挂起。这个动作把 Vim 停下来，并让你回到那个你启动 Vim 的命令外壳里。接着，你可以执行任何别的命令直到你觉得无聊为止。然后再用 "fg" 命令回到 Vim。

```
CTRL-Z
{任何外壳命令序列}
fg
```

你正好回到那个当初你离开 Vim 的地方，什么也没改变。
　当你按 CTRL-Z 行不通时，你还可以用命令 ":suspend"。别忘了把 Vim 引导回前台，否则你会失去你所有的修改！

只有 Unix 支持这个功能。在其它系统上 Vim 将为你启动一个外壳。这也让你得以执行外壳命令。可那是一个新外壳，而非你在其中启动 Vim 的那个。
　当你运行图像用户界面时，你无法返回那个 Vim 从其中启动的外壳。CTRL-Z 的作用只是把 Vim 窗口最小化。

21.2　执行外壳命令

从 Vim 内部执行单个外壳命令，用 ":!{command}"。例如，要显示目录表：

```
:!ls
:!dir
```

第一行用在 Unix 上，第二行用于微软视窗。
Vim 将执行该程序。当它结束时，你会得到提示，请按 <Enter> 键。这提示允许你在回去编辑你的文本之前看一看该命令的输出。
字符 "!" 也用在其它有个程序被调用运行的场合。让我们看一看共有哪些：

```
:!{program}             执行 {program}
:r !{program}           执行 {program} 并读取其输出
:w !{program}           执行 {program} 传送文本至其输入
:[range]!{program}      经由 {program} 过滤文本
```

注意 "!{program}" 前面那个作用区产生的区别可大了。不附带作用区，这个程序就跟通常一样被执行，而加了这个作用区，作用区内的文本行就经由该程序过滤而出。

用这种方法执行一系列命令也是可以的。但外壳在此却更胜一筹。你可以用这种方法启动一个新外壳：

```
:shell
```

这有点类似于用 CTRL-Z 来挂起 Vim。不同之处在于这种方法启动的是一个新外壳。

使用图像用户界面时，这个外壳利用 Vim 的窗口作为其输入和输出端口。既然 Vim 并非终端仿真器，所以使用中未必尽善尽美。要是你遇到麻烦，试试切换 'guipty' 选项。如果这方法仍不奏效，那就只好启动一个新的终端来运行外壳。例如：

```
:!xterm&
```

21.3 记忆有关信息；viminfo

在你编辑了一会儿文件以后，你就会有些文本储存在寄存器内，有些标记指向各种各样文件，还有一些精妙的命令保存在命令行历史记录内。当你退出 Vim，所有这些就全没了。但不用担心，你能够把它们找回来！

信息文件 viminfo 设计用来储存状态信息：

```
命令行和模式搜索的历史记录
寄存器内文本
各种文件的标记
缓存器列表
全局变量
```

你每次退出 Vim，它就把此种信息存放在一个文件内。即 viminfo 信息文件。当 Vim 重新启动时，就读取这个信息文件，而那些信息就被还原了。

选项 'viminfo' 的默认设定是还原有限的几种信息。你也许希望设定它记住更多的信息。下面这个命令可以办到：

```
:set viminfo=string
```

其中字符串 "string" 规定了什么要储存。该字符串的语法为一个选项字符跟一个参数。选项和参数组成的对子之间由逗号分隔。
来看一下你可以怎样构建你自己的 viminfo 字符串。首先，选项 ' 用于规定你为多少个文件保存标记 (a-z)。为此选项凑一个整数就行 (比如 1000)。你的命令现在看起来像这样：

```
:set viminfo='1000
```

选项 f 控制是否要储存全局标记 (A-Z 和 0-9)。如果这个选项设为 0，那么什么也不存储。如果设为 1，或你对 f 选项不作规定，那么标记就被存储。你要这个功能，现在你有了：

```
:set viminfo='1000,f1
```

选项 < 控制着每个寄存器内保存几行文本。默认情况下，所有的文本行都被保存。如果设为 0，则什么也不保存。为了避免成千上万行文本被加入你的信息文件 (那些文本可能永远也没用，徒然使 Vim 启动得更慢)，你采用 500 行的上限：

```
:set viminfo='1000,f1,<500
```

你也许用得着的其它选项：

```
:       保存命令行历史记录内的行数
@       保存输入行历史记录内的行数
/       保存搜索历史记录内的行数
r       可移动介质，其上的文件不保存标记 (可用多次)
!       以大写字母开头并且不含有小写字母的全局变量
h       启动时关闭选项 'hlsearch' 高亮显示
%       缓冲区列表 (只有当不带参数启动 Vim 时才用来恢复)
c       用编码 'encoding' 转换文本
n       用于 viminfo 文件的名称 (必须为最后一项选项)
```

欲知更多详情，参见 'viminfo' 选项以及 viminfo-file 。

当你多次运行 Vim，最后退出的那个就把信息储存起来。这可能导致以前退出的那些 Vim 所存放的信息流失，因为每个条目只能被记住一次。

重 返 VIM 中 断 处

你编辑一个文件到一半，但你得下班去度假了。你退出 Vim 就去享受你的人生，把你那些工作忘得干干净净。两个星期以后你启动 Vim，键入：

```
'0
```

你正好回到当初离开 Vim 的地方。所以你就把你的工作接着做下去。

你每次退出 Vim，它都创建一个标记。最后那个是 '0。原来那个 '0 所指的位置就成了 '1。而原来那个 '1 就成了 '2，依此类推。而原来的标记 '9 就没了。

要发现标记 '0 至 '9 指向何处， :marks 命令很有用。

重 返 某 文 件

如果你想回到最近曾经编辑过的文件，但已经退出过 Vim，有一个稍微复杂的方法。你可以这样看到这些文件的列表：

```
:oldfiles
1: ~/.viminfo
2: ~/text/resume.txt
3: /tmp/draft
```

假如你要编辑第二个文件，也就是列表里 "2:" 开始的那个，输入：

```
:e #<2
```

其它接受文件名参数的命令都能替代这里的 ":e"，"#<2" 这种形式可以用在 "%"（当前文件名）和 "#"（轮换文件名）可以出现的地方。由此，你也可以这样来分割窗口来编辑第三个文件：

```
:split #<3
```

#<123 这种东西在你要用来编辑文件时毕竟是太麻烦了。幸运的是有个简单点的办法：

```
:browse oldfiles
1: ~/.viminfo
2: ~/text/resume.txt
3: /tmp/draft
-- More --
```

你可以得到和 :oldfiles 相同的文件。如果要编辑 "resume.txt"，先按 "q" 停止列表，然后会有提示：

```
Type number and <Enter> (empty cancels):
```

输入 "2" 并按 <Enter> 来编辑第二个文件。

如果知道文件名包含的模式，也可 :filter 文件列表：

```
:filter /resume/ :browse oldfiles
```

因为只有一个匹配的文件名，Vim 会直接编辑该文件而不用提示。如果过滤器匹配多个文件，则会给出提示，从匹配文件列表中选择：

```
:filter! /resume/ browse oldfiles
1: ~/.viminfo
3: /tmp/draft
Type number and <Enter> (q or empty cancels):
```

注意：这次我们过滤掉所有 不 匹配 resume 的文件。

详见 :oldfiles 、 v:oldfiles 和 c_#< 。

VIM 间 信 息 移 动

在 Vim 仍然运行的情况下，你可以用命令 ":wviminfo" 和 ":rviminfo" 来保存和还原信息。这很方便，比如在两个同时运行的 Vim 之间交换寄存器内容。在第一个 Vim 里执行：

```
:wviminfo! ~/tmp/viminfo
```

而在第二个 Vim 里执行：

```
:rviminfo! ~/tmp/viminfo
```

很明显，字符 "w" 指 "写" 而字符 "r" 指 "读"。
":wviminfo" 用字符 ！ 来强制重写一个已存在文件。如果省略而文件却存在，那么这些信息就跟那个文件合并到一起。
用于 ":rviminfo" 的 ！ 字符意味着所有的信息都被采用，这可能会重写一些已存在信息。若不用 ！ 字符，则只有那些尚未设定的信息才会被采用。
这些命令也可以用来储存信息为将来所用。你可以让一个目录专门存放信息文件，而每一个所包含的信息各有其特殊的目的。

21.4 会话

假定你编着编着，编到那一天要结束了。你想放下手上的工作，而于第二天再从你停下来的那地方继续编下去。你可以做到这一点，只要把你的编辑会话保存起来，第二天再把它还原回来。
Vim 会话存放着所有跟你的编辑相关的信息。这包括诸如文件列表、窗口布局、全局变量、选项、以及其它信息。(究竟什么信息被记住，则由选项 'sessionoptions' 控制，稍后叙述。)
下面这个命令创建一个会话文件:

```
:mksession vimbook.vim
```

如果你以后要还原这个会话，你可以用这个命令:

```
:source vimbook.vim
```

如果你要启动 Vim 并还原某个特别的会话，你可以用下面这个命令:

```
vim -S vimbook.vim
```

这命令告诉 Vim 在启动时读取一个特定的会话文件。参数 'S' 指会话 (实际上，你可以用 -S 运行任何 Vim 脚本，因而，你也不妨指其为运行脚本，"source")。

那个曾经打开的窗口就还原了，跟以前一样的位置和大小。映射和选项值也像以前一样。
究竟还原了什么取决于 'sessionoptions' 选项。默认值为 "blank,buffers,curdir,folds,help,options,tabpages,winsize,terminal"。

blank	保留空窗口
buffers	所有缓冲区，而非仅仅一个窗口内的
curdir	当前目录
folds	折叠，包括人工创建的
help	帮助窗口
options	所有选项和映射
tabpages	所有标签页
winsize	窗口大小
terminal	包含终端窗口

你爱怎么改就怎么改。例如，除了上述项目以外，你还要还原 Vim 窗口大小:

```
:set sessionoptions+=resize
```

会 话 用 法

最显而易见的会话用法，是在编辑属于不同项目的文件时。假定你把会话文件都储存在目录 "~/.vim" 下。你正编辑着项目 "secret" 的文件，而你必须切换到项目 "boring" 的文件上：

```
:wall
:mksession! ~/.vim/secret.vim
:source ~/.vim/boring.vim
```

首先用命令 ":wall" 把所有修改过的文件存盘。然后用命令 ":mksession!" 保存当前会话。它重写了前一次会话文件。下一次载入 "secret" 会话时，你便可以在原来那一点上接着编辑下去。最后，你载入新的 "boring" 会话。

如果你打开帮助窗口，分割和关闭各种各样窗口，或者一般来说把窗口布局搞砸了，此时你可以恢复上次保存的会话：

```
:source ~/.vim/boring.vim
```

你因而享有充分的控制权，要么把当前会话中的设置保存起来，以便下一次编辑时从现在这个位置接着编下去，要么保留会话文件不变，一切都从那儿开始。

另一种会话用法是你创建了一种你喜欢的窗口布局，并把它保存在一个会话文件中。然后你可以在任何时候恢复这种窗口布局。

例如，这是一个使用起来很不错的布局：

```
+----------------------------------------+
|                  VIM - main help file  |
|                                        |
|Move around:  Use the cursor keys, or "h|
 help.txt================================
|explorer   |                            |
|dir        |~                           |
|dir        |~                           |
|file       |~                           |
|file       |~                           |
|file       |~                           |
|file       |~                           |
 ~/========= [No File]==================|
|                                        |
+----------------------------------------+
```

其顶部有个帮助窗口，使你能够阅读本文。左边那个狭长窗口相当于一个文件浏览器。这是一个 Vim 插件，用来表列一个目录的内容。你可以在那儿挑选文件来编辑。有关这一点，下一章有更多叙述。

从一个刚刚启动的 Vim 创建这样一个布局，请用：

```
:help
CTRL-W w
:vertical split ~/
```

你可以根据你的喜好稍稍改动一点那些窗口的大小。然后保存会话记录：

```
:mksession ~/.vim/mine.vim
```

现在你可以用这个布局启动 Vim：

```
vim -S ~/.vim/mine.vim
```

提示：要在一个空窗口中打开一个表列在浏览器窗口中的文件，请把光标移到文件名上并按 "O"。用鼠标双击也行。

UNIX 和 微 软 视 窗

有些人不得不今天在微软视窗上，而明天则在 Unix 上工作。如果你是其中之一，请考虑把 "slash" 和 "unix" 加入选项 'sessionoptions'。此后存盘的会话文件的格式在两种系统上都适用。请把以下命令加进你的 vimrc 文件内：

```
:set sessionoptions+=unix,slash
```

Vim 将因而使用 Unix 格式，因为微软视窗上的 Vim 能读写 Unix 文件，但 Unix 上的 Vim 却不能读取微软视窗格式的会话文件。类似地，微软视窗上的 Vim 懂得文件路径名称里用以分隔名字的 / ，但 Unix 上的 Vim 却不懂微软视窗上的 \。

会 话 记 录 和 信 息 文 件

会话记录储存了不少东西，但不储存标记位置，寄存器内容以及命令行历史记录。你需要利用 Vim 信息文件 viminfo 储存这些信息。

在大多数情况下，你将需要利用会话记录而非全部信息文件内的信息。这么做可以让你切换到另一个会话记录，但却保留着命令行历史记录。并得以在一个会话期内把文本抄进寄存器，而于另一个会话期把它粘贴出来。

你也许宁可用会话记录保存信息。那么，你必须亲自动手。例如：

```
:mksession! ~/.vim/secret.vim
:wviminfo! ~/.vim/secret.viminfo
```

而再次把它还原：

```
:source ~/.vim/secret.vim
:rviminfo! ~/.vim/secret.viminfo
```

21.5 视图

会话记录储存着整个 Vim 窗口外观。当你只需要为某个窗口储存特性时，得用视图。

视图的用处在于你要以某种特定的方式编辑一个文件。例如，你以 'number' 选项显示了行号，并定义了若干折叠。正如会话记录那样，你可以记住这一视图并在以后还原回来。事实上，当你储存会话时，每个窗口的视图都储存了。

视图有两种基本用法。第一种是让 Vim 为视图文件挑一个文件名。你可以在以后编辑同一文件时还原该视图。为当前窗口储存视图，用：

```
:mkview
```

Vim 将自行决定视图的储存位置。当你以后编辑同一文件时，用这个命令恢复该视图：

```
:loadview
```

这挺容易，不是吗？

现在你要阅读这个文件，阅读时你不要 'number' 选项显示行号，或者你要把所有的

折叠都打开。你可以设定这些选项，使窗口看起来就是你要的那个样子。然后储存这个视图：

```
:mkview 1
```

显而易见，你可以用下面的命令把它恢复：

```
:loadview 1
```

现在你可以用 ":loadview" 在这个文件的两个视图之间切换了，一个加参数 "1"，另一个则不加参数。

你可以用这个方法为同一个文件储存视图达十个之多，一个没序号的，而九个则有序号 1 至 9。

命 名 视 图

第二种视图基本用法是把视图储存在一个你选定的文件内。你可以在编辑另外一个文件时载入这个视图。Vim 将接着转而编辑该视图规定的文件。这样你就可以用这种方法迅速换个文件来编辑，其所有的选项设定就跟它们在存盘时一样。

例如，要保存当前文件的视图：

```
:mkview ~/.vim/main.vim
```

你可以用这个命令把它还原：

```
:source ~/.vim/main.vim
```

21.6 模式行

当你编辑一个特定的文件，你也许为该文件设定了特定的选项。每次键入这些命令很无聊。而在许多人共享一个文件时，利用会话和视图来编辑这个文件也无济于事。

解决这个困境的方法是给文件加一个模式行。那是一行文本，它把一些只适用于该文件的选项设定告诉 Vim。

一个典型的例子是在一个 C 程序中你把缩进值设为 4 的倍数。这就要求把选项 'shiftwidth' 设为 4。这个模式行能奏效：

```
/* vim:set shiftwidth=4: */
```

把这一行插入该文件起首五行或结尾五行。编辑这个文件时，你将注意到 'shiftwidth' 选项已经设定为 4。编辑另一个文件时，它再设回默认值 8。

对于有些文件，模式行放在头部挺合适，所以它应该置于该文件的顶部。对于文本文件和其它那些模式行会影响正文阅读的文件，把模式行放在文件结尾处。

选项 'modelines' 规定了要在文件起首和结尾几行之内检查那儿是否包含了模式行。要检查十行：

```
:set modelines=10
```

选项 'modeline' 可以用来撤销这个设定。如果你以超级用户身份 (Unix 的 root 或 MS-Windows 的 Administrator) 工作或者你不信任要编辑的文件时应该如此：

```
:set nomodeline
```

模式行可以用这种格式：

```
any-text vim:set {option}={value} ... : any-text
```

其中 "any-text" 表示你可以在 Vim 实际用到的部分之前和之后加任意文本。这就允许你使它看起来像个注释，正如上例采用了 /* 和 */。
" vim:" 部分使 Vim 识别出这个模式行。在 "vim" 的前面必须有空格，除非 "vim" 置于行首。因此像 "gvim:" 这样用法是不行的。
冒号之间的部分是命令 ":set" 。它的用法就跟键入 ":set" 命令一模一样，只除了在命令包含的任何冒号之前需要插入一个反斜杠（否则该冒号就会被看成模式行结尾）。

还有一个例子：

```
// vim:set textwidth=72 dir=c\:\tmp:  use c:\tmp here
```

在第一个冒号之前多一个反斜杠，因此它被包括在 ":set" 命令内。第二个冒号后面的文本则被忽略不计，因此可以在那儿放个注释。

欲知详情，参见 modeline 。

下一章： usr_22.txt 寻找要编辑的文件

VIM 用户手册 - by Bram Moolenaar
译者：Chimin Yen

寻找要编辑的文件

到处都是文件，你怎么查找它们呢？Vim 为我们提供了在目录树间浏览的种种方法。还有若干命令让你从一个文件通过文件名跳转到一个文件。而且，Vim 还记着曾经编辑过哪些文件。

22.1　文件浏览器

Vim 有个插件可以用来编辑一个目录。试一下这个命令：

```
:edit .
```

借助魔术般的自动命令和 Vim 脚本功能，目录的内容被用来填充窗口。看起来就像这样：

```
" ============================================================================
" Netrw Directory Listing                                        (netrw v109)
"   Sorted by      name
"   Sort sequence: [\/]$,\.h$,\.c$,\.cpp$,*,\.info$,\.swp$,\.o$\.obj$,\.bak$
"   Quick Help: <F1>:help  -:go up dir  D:delete  R:rename  s:sort-by  x:exec
" ============================================================================
../
./
check/
Makefile
autocmd.txt
change.txt
eval.txt
filetype.txt
help.txt.info
```

你可以看到这样一些条目：

1. 浏览工具的名称和版本号
2. 浏览目录名
3. 排序方法（可以是名字、时间或大小）

4. 名字如何排序（目录优先，然后是 *.h 文件、*.c 文件，等等）
5. 如何得到帮助（用 <F1> 键），然后是可用命令的概括列表
6. 文件列表，包括 "../"，用户可以由此列出父目录。

如果你启用了语法高亮功能，那么，目录里不同部分就显示成不同颜色，让你比较容易辨认它们。

你可以用普通模式下的 Vim 命令在文本内到处移动。例如，移动光标到一个文件名上并按下 <Enter> 键。你就可以编辑那个文件了。要回到浏览器，再用一次 ":edit ." 或 ":Explore" 即可，按 CTRL-O 也行。

试一下把光标移至某个目录名，按 <Enter> 键。结果，浏览器就进了该目录，并把那里的条目显示出来。对准第一个目录 "../" 按 <Enter> 键， 让你返回父目录。按 "-" 键可达到同样的目的，且无须先把光标移到 "../" 条目上。

你可以按 <F1> 键获取关于 netrw 文件浏览器功能的帮助。帮助文字如下：

```
    9. Directory Browsing        netrw-browse   netrw-dir   netrw-list   netrw-help

    MAPS                                                                  netrw-maps
         <F1>.............Help.....................................|netrw-help|
         <cr>.............Browsing..................................|netrw-cr|
         <del>............Deleting Files or Directories.............|netrw-delete|
         -................Going Up.................................|netrw--|
         a................Hiding Files or Directories...............|netrw-a|
         mb...............Bookmarking a Directory...................|netrw-mb|
         gb...............Changing to a Bookmarked Directory.........|netrw-gb|
         cd...............Make Browsing Directory The Current Dir....|netrw-c|
         d................Make A New Directory......................|netrw-d|
         D................Deleting Files or Directories.............|netrw-D|
         <c-h>............Edit File/Directory Hiding List...........|netrw-ctrl-h|
         i................Change Listing Style......................|netrw-i|
         <c-l>............Refreshing the Listing....................|netrw-ctrl-l|
         o................Browsing with a Horizontal Split..........|netrw-o|
         p................Use Preview Window........................|netrw-p|
         P................Edit in Previous Window...................|netrw-p|
         q................Listing Bookmarks and History.............|netrw-qb|
         r................Reversing Sorting Order...................|netrw-r|
        (等等)
```

<F1> 键把你带到 netrw 目录浏览内容的帮助页面。这是一个常规的帮助页面，同样常规的 CTRL-] 跳转到带标签的帮助项目，而 CTRL-O 则返回。

要选择显示和编辑的文件：(光标在某文件名上)

<enter>	在当前窗口打开文件	netrw-cr
o	横向分割窗口并显示文件	netrw-o
v	竖向分割窗口并显示文件	netrw-v
p	使用 preview-window	netrw-p
P	在上次的窗口中编辑	netrw-P
t	在新标签页中打开文件	netrw-t

接下来的普通模式命令用来控制浏览器显示：

i	控制列表风格（瘦、长、宽和树形)。长列表包含文件大小和日期信息。
s	反复按 s 会改变文件排序的方式；可以按照名字、修改日期或文件大小排序。
r	逆转排列顺序。

略举数例其它的普通模式命令:

cd	把当前目录改成显示在窗口中的那个目录。(见 g:netrw_keepdir ，它也控制此行为)
R	为光标下文件改名。Vim 将提示你提供新文件名。
D	删除光标下文件。Vim 将提示你确认删除动作。
mb gb	建立书签/转到书签

还有命令模式；还是一样，只举数例:

:Explore [directory]	浏览指定/当前目录
:NetrwSettings	当前 netrw 设置的综合列表，带有帮助链接。

netrw 浏览器不限于你的本地机器；也可使用 url，如下:　(拖尾的 / 是必需的)

```
:Explore ftp://somehost/path/to/dir/
:e scp://somehost/path/to/dir/
```

详见 netrw-browse 。

22.2 当前目录

正如外壳一样，Vim 也有当前目录的概念。假设你在主目录，并要编辑几个储存在目录 "VeryLongFileName" 下的文件。你可以这样做:

```
:edit VeryLongFileName/file1.txt
:edit VeryLongFileName/file2.txt
:edit VeryLongFileName/file3.txt
```

为了避免太多的键击，你可以这样做:

```
:cd VeryLongFileName
:edit file1.txt
:edit file2.txt
:edit file3.txt
```

":cd" 命令可以用来改变当前目录。你可以用 ":pwd" 命令来查看当前目录使什么:

```
:pwd
/home/Bram/VeryLongFileName
```

Vim 记得最近访问过的那个目录。你可以用 "cd -" 命令回去那儿。例如:

```
:pwd
/home/Bram/VeryLongFileName
:cd /etc
```

```
:pwd
/etc
:cd -
:pwd
/home/Bram/VeryLongFileName
:cd -
:pwd
/etc
```

窗口本地目录

当你把窗口一分为二以后，两个窗口的当前目录是一样的。而当你要在新窗口内编辑几个存放在当前目录以外某处的文件时，你可以让新窗口采用不同的当前目录，同时保持原窗口当前目录不变。新窗口采用的当前目录称为本地目录。

```
:pwd
/home/Bram/VeryLongFileName
:split
:lcd /etc
:pwd
/etc
CTRL-W w
:pwd
/home/Bram/VeryLongFileName
```

只要你不发出 :lcd 命令，所有窗口共享同一个当前目录。在一个窗口执行一次 :cd 命令，也同时改变其它窗口的当前目录。

执行过 :lcd 命令的窗口记得它特有的当前目录。在其它窗口执行 :cd 或 :lcd 命令对它毫无影响。

在一个采用特有当前目录的窗口执行 :cd 命令以后，该窗口就回过头来又采用共享的当前目录了。

标签页本地目录

打开新的标签页时，使用打开时所在的之前标签页窗口的目录。可以用 :tcd 命令来改变当前标签页的目录。除了有局部于窗口的目录的窗口以外，同一标签页的所有窗口共享此目录。任何此标签页上打开的新窗口会使用此目录为当前工作目录。在标签页中使用 :cd 命令不会改变有局部于标签页目录的标签页的工作目录。在标签页中使用 :cd 命令改变全局目录后，也改变了当前标签页的工作目录。

22.3 查找文件

假定现在你在编辑一个 C 程序，该程序有这样一行:

```
#include "inits.h"
```

你想要查看文件 "inits.h" 里有些什么。把光标移到该文件名上，并键入:

```
gf
```

Vim 就会找到并打开这个文件。
　那么，如果该文件不在当前目录里怎么办？Vim 将利用 'path' 选项来寻找这个文件。该选项是一系列目录名，Vim 会在其中为你寻找文件。
　假设你把你的头文件存放在目录 "c:/prog/include" 里。下面这个命令把该目录加入 'path' 选项：

```
:set path+=c:/prog/include
```

这个目录名是一绝对路径。不管你在哪儿，它都指向同一目录。如果你已经知道要编辑的文件位于当前文件所在目录下某个子目录里，你该怎么办呢？那样的话，你可以指定一个相对路径。相对路径以 "." 开始：

```
:set path+=./proto
```

这个命令告诉 Vim 到目录 "proto" 里找文件，而 "proto" 则位于你在其中用了 "gf" 命令的文件所在的目录下面。如此一来，对 "inits.h" 用 "gf" 命令，就让 Vim 以当前文件所在目录为起点，寻找 "proto/inits.h"。
　如果不加 "./"，只用 "proto"，Vim 就会进入当前目录下的 "proto" 目录寻找。然而当前目录可能不是你在编辑的这个当前文件所在的目录。

'path' 选项还允许你用其它许多方法来指定在其中寻找文件的目录。参见关于 'path' 选项的帮助。
　'isfname' 选项用来决定哪些字符可用于文件名，以及哪些不可以（如上例中的双引号 " 字符）。

当你要找的文件名没出现在你编辑的文件里时，你可以键入这个文件名：

```
:find inits.h
```

然后 Vim 就利用 'path' 选项来确定该文件的位置。这就跟 ":edit" 命令一样，只不过 ":edit" 命令不用 'path' 选项。

要在新窗口内打开那个已经找到的文件，用 CTRL-W f，而不是 "gf"，或用 ":sfind" 而不是 ":find"。

有个好办法可以直接启动 Vim 来编辑 'path' 中的某个文件：

```
vim "+find stdio.h"
```

这会在你的 'path' 中查找 "stdio.h"。双引号是必需的，用来把括起来的当做一个参数 -+c 。

22.4　缓冲区列表

Vim 编辑器使用术语 "缓冲区" 来描述编辑当中的文件。实际上，缓冲区是你编辑的文件的副本。你修改完缓冲区，就把缓冲区的内容写进文件。缓冲区不仅存放文件内容，而且还存放着全部标记，设定，以及其它跟被编辑文件相关的东西。

隐藏的缓冲区

假设你在编辑文件 one.txt，同时又要编辑文件 two.txt。你本来可以简单地用 ":edit two.txt" 来办到，但由于你已经修改了 one.txt，那样做就没用了。而你又不想在此时

就把 one.txt 存盘。Vim 可以为你解决这个问题:

```
:hide edit two.txt
```

缓冲区 "one.txt" 从屏幕消失，但 Vim 仍然知道你在编辑这个缓冲区，因而保留着它修改过的文本。这样的缓冲区称为隐藏的缓冲区：缓冲区存放着文本，但你看不见它。

":hide" 命令参数是另一个命令。":hide" 使得那个命令表现得就像 'hidden' 选项已被设定。你也可以不用 ":hide" 命令而设定 'hidden' 选项。其作用是当离开任何缓冲区时，该缓冲区变成隐藏。

小心！当你的隐藏的缓冲区已经改动，千万不要在所有缓冲区存盘之前就退出 Vim。

非 激 活 缓 冲 区

一个缓冲区一经使用，Vim 就记住了一些有关该缓冲区的信息。即使它既不显示在窗口内，又非隐藏缓冲区，它也仍然在缓冲区列表上。这样的缓冲区称为非激活缓冲区。一般而言，

激活	显示在窗口内，并加载文本
隐藏	不显示在窗口内，但加载文本
非激活	不显示在窗口内，不加载文本

非激活缓冲区不会被遗忘，因为 Vim 保存着关于它们的信息，如标记等。而且记住文件名有个好处，你可以调阅你编辑过的文件名，再次编辑它们。

缓 冲 区 列 表

你可以用这个命令查看缓冲区列表:

```
:buffers
```

另一个作用相当的命令，虽然意思不那么明白，但键入时省事多了:

```
:ls
```

其输出可能像这样:

```
  1 #h   "help.txt"                    line 62
  2 %a + "usr_21.txt"                  line 1
  3      "usr_toc.txt"                 line 1
```

第一栏存放缓冲区号。你可以利用它来编辑文件，而不用键入文件名，参见下文。

紧随缓冲区号的是些标志位。然后是文件名，以及光标最后一次停留的行号。

可能出现的标志位有以下这些 (自左至右):

u	列表外缓冲区 unlisted-buffer 。
%	当前缓冲区。
#	轮换缓冲区。
a	激活缓冲区，缓冲区被加载且显示。
h	隐藏缓冲区，缓冲区被加载但不显示。
=	只读缓冲区。
-	不可改缓冲区，'modifiable' 选项不置位。
+	已修改缓冲区。

编 辑 缓 冲 区

你可以通过指定其缓冲区号来编辑一个缓冲区，而不必键入文件名：

```
:buffer 2
```

但获知缓冲区号的唯一途径是查阅缓冲区列表。如果不用缓冲区号，你可以用文件名，或其部分：

```
:buffer help
```

Vim 将为你键入的文件名找到最佳匹配。如果只有一个缓冲区与之匹配，该缓冲区就被选用。在这个例子中，被选中的就是 "help.txt"。
要在新窗口中打开一个缓冲区：

```
:sbuffer 3
```

这方法也适用于文件名。

使 用 缓 冲 区 列 表

你可以用这些命令在缓冲区列表间移动：

```
:bnext          编辑下一个缓冲区
:bprevious      编辑前一个缓冲区
:bfirst         编辑第一个缓冲区
:blast          编辑最后一个缓冲区
```

要从缓冲区列表上删除一个缓冲区，用这个命令：

```
:bdelete 3
```

同样，这命令也适用文件名。
如果你删除了一个激活的缓冲区（显示在窗口中的缓冲区），你也就关闭了该窗口。如果你删除了当前缓冲区，你也就关闭了当前窗口。如果它是最后一个窗口，Vim 将找一个缓冲区来编辑。你无法什么也不编辑！

备注：
即使用 ":bdelete" 命令删除了缓冲区以后，Vim 依然记得它。这个缓冲区实际上成了 "列表外" 缓冲区，它不再出现在 ":buffers" 命令所报告的列表中。不过 ":buffers!" 命令仍会列出 "列表外" 缓冲区（没错，Vim 无所不能）。要让 Vim 彻底忘记一个缓冲区，用 ":bwipe" 命令。另见 'buflisted' 选项。

下一章： usr_23.txt 编辑其它文件

usr_23.txt　　适用于 Vim 9.1 版本。　　最近更新：2020年12月

VIM 用户手册 - by Bram Moolenaar
译者：Chimin Yen

编辑特殊文件

本章讨论特殊文件的编辑。利用 Vim 你可以编辑经过压缩或加密的文件。有些文件需要在互联网上存取。二进制文件也能有限度地编辑。

下一章：　usr_24.txt　　快速插入
前一章：　usr_22.txt　　寻找要编辑的文件
目录：　usr_toc.txt

23.1　DOS、Mac 和 Unix 文件

很久以前，老式的电传打字机使用两个字符来另起新行。一个字符把滑动架移回首位（称为回车，<CR>），另一个字符把纸上移一行（称为换行，<LF>）。

当计算机问世以后，存储器曾经非常昂贵。有些人就认定没必要用两个字符来表示行尾。UNIX 开发者决定他们可以用 <New Line> 或 <NL> 一个字符来表示行尾。Apple 开发者规定了用 <CR>。开发 Microsoft Windows 的那些家伙则决定沿用老式的 <CR><NL>（在帮助文本我们使用 <NL> 代表换行符）。

那意味着，如果你试图把一个文件从一种系统移到另一种系统，那么你就有换行符方面的麻烦。Vim 编辑器自动识别不同文件格式，并且不劳你操心就把事情给办妥了。

选项 'fileformats' 包含各种各样的格式，Vim 会在编辑一个新文件之初尝试该选项定义的各种格式。例如，下面这个命令告诉 Vim 先尝试用 UNIX 格式，其次，尝试 MS-DOS 格式：

```
:set fileformats=unix,dos
```

编辑一个文件时，你将注意到 Vim 给出的信息中包括文件所用的格式。如果你编辑的是本地格式文件（你编辑的文件格式和所用系统一致），你就不会看到任何格式名。因此在 Unix 系统上编辑一个 Unix 格式文件不会产生任何关于格式的信息。但你若编辑一个 dos 文件，Vim 将这样通知你：

```
"/tmp/test" [dos] 3L, 71C
```

如果是 Mac 文件，你会看到 "[mac]"。

探测到的文件格式会被存入 'fileformat' 选项。执行下面这个命令可以显示你当前使用的文件格式：

```
:set fileformat?
```

Vim 能使用的三种格式如下：

```
unix            <NL>
dos             <CR><NL>
mac             <CR>
```

使 用 MAC 格 式

在 Unix 上，<NL> 用于分行。但 <CR> 字符混在文本行中间也非罕见。这种情况碰巧经常发生在 Vi (和 Vim) 脚本内。

在采用 <CR> 作为换行符的 Macintosh 上，<NL> 字符也有可能混在文本行中间。

结果，很难 100% 肯定一个同时包含 <CR> 和 <NL> 的文件究竟是 Mac 还是 Unix 格式。所以，Vim 假设你一般不会在 Unix 上编辑一个 Mac 文件，所以干脆对这种文件格式不作检查。果真要检查此种格式，就把 "mac" 加入 'fileformats':

```
:set fileformats+=mac
```

然后 Vim 就会猜测文件格式。要当心，Vim 可能会猜错的。

强 制 格 式

如果你用往日美好的 Vi 来尝试编辑一个采用 MS-DOS 格式的文件，你将会发现每一行的末尾有个 ^M 字符。(^M 就是 <CR>)。而 Vim 的自动探测功能就避免了这个问题。莫非你确实要按那个样子来编辑这个文件吗？那么你需要强制 Vim 忽略文件格式而使用你指定的格式:

```
:edit ++ff=unix file.txt
```

字符串 "++" 告诉 Vim 后面跟的是选项名，以取代其默认值。但仅作用于这一个命令。"++ff" 用于 'fileformat' 选项。你也可以用 "++ff=mac" 或 "++ff=dos"。

这样用法并非适用于任意选项，目前 Vim 仅仅实现了 "++ff" 和 "++enc"。用全称 "++fileformat" 和 "++encoding" 也行。

转 换

你可以用 'fileformat' 选项把文件从一种格式转换为另一种。例如，假定你有个名为 README.TXT 的 MS-DOS 文件，你要把它转换成 UNIX 格式。首先编辑这个采用 MS-DOS 格式的文件:

```
vim README.TXT
```

Vim 将识别出那是一个 dos 格式文件。现在把这个文件的格式改为 UNIX:

```
:set fileformat=unix
:write
```

这个文件就以 Unix 格式存盘了。

23.2 互联网上的文件

有人给你传送了一封电子邮件，其中引用了一个以超链接 URL 表示的文件。例如:

```
You can find the information here:
        ftp://ftp.vim.org/pub/vim/README
```

你当然可以启动一个程序来下载这个文件，把它存入你本地磁盘，然后启动 Vim 来编辑它。

但有一个简单得多的方法。把光标移到那个超链接里任何一个字符上，然后使用这个命令：

```
gf
```

运气好的话，Vim 将确定用哪个程序来下载这个文件，并把文件下载下来让你编辑该副本。在另一个新窗口打开这个文件，则用 CTRL-W f。

如果事情不顺利的话，你会得到出错信息。可能那个链接有错，也可能你没有阅读它的权限，还可能网络连接中断了，等等。不幸的是，很难搞清楚出错的原因。你也许得尝试以手工方法来下载这个文件。

访问因特网上的文件要安装插件 netrw 才行。目前，下面这些格式的超文本链接可被识别：

```
ftp://          使用 ftp
rcp://          使用 rcp
scp://          使用 scp
http://         使用 wget (只读)
```

Vim 并非亲自与因特网联系。它有赖于你的计算机上安装好的上面提及的程序。大多数 Unix 系统上装有 "ftp" 和 "rcp"。而 "scp" 和 "wget" 也许要另外安装。

无论你用哪个命令开始编辑，Vim 总是会探测一下这些超文本链接。包括如 ":edit" 和 ":split" 这样的命令。存盘命令也行，但除了 http:// 之外。

欲知详情，包括密码问题，参见 netrw 。

23.3 加密

有些信息你希望保留给自己。例如，当你在计算机上写一份给学生用的试卷。你不会愿意在考试开始之前给聪明的学生琢磨出一种偷阅试题的方法。Vim 能为你给文件加密，为你提供一些保护。

要开始编辑一个需要加密的新文件，可以用 "-x" 参数来启动 Vim。例如：

```
vim -x exam.txt
```

Vim 提示你确定一个密码，用于为文件加密和解密：

```
Enter encryption key:

(译者注: 中文是
输入密码:
)
```

现在仔细键入那个密码。你所键入的字符将为星号代替，因此你看不到。为了避免由于打字错误引起麻烦，Vim 要求你再输入一次密码：

```
Enter same key again:
```

```
(译者注: 中文是
请再输入一次:
)
```

现在你可以像平时一样编辑这个文件并把你所有的秘密放进去。当你编完文件要退出 Vim 时，这个文件就被加密存盘了。

当你用 Vim 编辑这个文件时，它就会要求你再输入那同一个密码。你不需要用 "-x" 参数。你也可以用普通的 ":edit" 命令编辑加密的文件。Vim 给这个文件加上特定的 magic 字符串，据以识别那是经过加密的文件。

如果你试图用另一个程序来阅读这个文件，你将读到一堆垃圾。如果你用 Vim 来编辑这个文件，但输入了错误的密码，你也只能得到垃圾。Vim 并不具备检验密码正确性的机理 (这一点使得破译密码更为困难)。

开 或 关 加 密

要给一个文件撤除加密，设定 'key' 选项为空字符串:

```
:set key=
```

你下次把这个文件存盘时，存盘的文件就是未经加密的。

设定 'key' 选项来启用加密是个坏主意。因为密码会被清清楚楚地显示在屏幕上。任何人都可以偷看到你的密码。

为了避免这样的问题，创造了 ":X" 命令。它会像 "-x" 参数向你索取一个密码:

```
:X
Enter encryption key: ******
Enter same key again: ******
```

加 密 的 局 限 性

Vim 采用的加密算法不够强大。它对于防止那种偷窥者是绰绰有余了，但不足以防止一个手上有大量时间的密码专家。交换文件和撤销文件中的文本也是加密的。不过，那是按块进行的，从而可能会缩短破解密码所需的时间。你可以关闭交换文件，但崩溃会丢失您的工作，因为 Vim 会只在内存里保留所有的文本。关闭撤销文件唯一的缺点是缓冲区卸载后不能撤销。

要避免使用交换文件，在命令行上用 -n 参数。例如，要编辑经过加密的文件 "file.txt"，但不用交换文件，请用下面的命令:

```
vim -x -n file.txt
```

如果你已经在编辑这个文件了，那么交换文件 swapfile 可以用下面的命令禁止:

```
:setlocal noswapfile
```

由于没了交换文件，文件复原就不可能了。为了避免失去编辑的成果，要比平时更勤快地存盘你的文件。

文件在内存中以明文形式存在。因此任何具备权限的人都能进入编辑者的内存浏览，从而，发现这个文件的内容。

如果你使用信息文件 viminfo，别忘了文本寄存器的内容也是明明白白写在其中的。

如果你真的要保证一个文件内容的安全，那么，你必须永远只在一个不联网的可携式计算机上编辑这个文件，使用优良的加密工具，并且在不用时，把你的计算机锁进一个大保险箱。

23.4 二进制文件

你可以用 Vim 来编辑二进制文件。Vim 本非为此而设计的，因而有若干局限。但你能读取一个文件，改动一个字符，然后把它存盘。结果是你的文件就只有那一个字符给改了，其它的就跟原来那个一模一样。

要保证 Vim 别把它那些聪明的窍门用错地方，启动 Vim 时加上 "-b" 参数:

```
vim -b datafile
```

这个参数设定了 'binary' 选项。其作用是排除所有的意外副作用。例如，'textwidth' 设为零，免得文本行给擅自排版了。并且，文件一律以 Unix 文件格式读取。

二进制模式可以用来修改某程序的消息报文。小心别插入或删除任何字符，那会让程序运行出问题。用 "R" 命令进入替换模式。

文件里的很多字符都是不可显示的。用 Hex 格式来显示它们的值:

```
:set display=uhex
```

另外，也可以用命令 "ga" 来显示光标下的字符值。当光标位于一个 <Esc> 字符上时，该命令的输出看起来就像这样:

```
<^[>  27,  Hex 1b,  Octal 033
```

文件中也许没那么多换行符。你可以关闭 'wrap' 选项来获得总览的效果:

```
:set nowrap
```

字 节 位 置

要发现你在文件中的当前字节位置，请用这个命令:

```
g CTRL-G
```

其输出十分冗长:

```
Col 9-16 of 9-16; Line 277 of 330; Word 1806 of 2058; Byte 10580 of 12206
```

最后两个数字就是文件中的当前字节位置和文件字节总数。这已经考虑了 'fileformat' 选项导致换行符字节不同的影响。

要移到文件中某个指定的字节，请用 "go" 命令。例如，要移到字节 2345:

```
2345go
```

使 用 XXD

一个真正的二进制编辑器用两种方式来显示文本：二进制和十六进制格式。你可以在 Vim

里通过转换程序 "xxd" 来达到这效果。该程序是随 Vim 一起发布的。
首先以二进制方式编辑这个文件：

```
vim -b datafile
```

现在用 xxd 把这个文件转换成十六进制：

```
:%!xxd
```

文本看起来像这样：

```
0000000: 1f8b 0808 39d7 173b 0203 7474 002b 4e49  ....9..;..tt.+NI
0000010: 4b2c 8660 eb9c ecac c462 eb94 345e 2e30  K,.`.....b..4^.0
0000020: 373b 2731 0b22 0ca6 c1a2 d669 1035 39d9  7;'1."......i.59.
```

现在你可以随心所欲地阅读和编辑这些文本了。 Vim 把这些信息当作普通文本来对待。修改了十六进制部分并不导致可显示字符部分的改变，反之亦然。
最后，用下面的命令把它转换回来：

```
:%!xxd -r
```

只有十六进制部分的修改才会被采用。右边可显示文本部分的修改忽略不计。

欲知更多详情，参见 xxd 手册。

23.5 压缩文件

这很容易：你可以像编辑任何其它文件一样，来编辑一个经过压缩的文件。插件 "gzip" 负责在你编辑这个文件时把它解压缩，以及在你存盘时再把它压缩起来。
目前支持的压缩方法有下面这些：

```
.Z      compress
.gz     gzip
.bz2    bzip2
```

Vim 调用上面提到的程序来实际完成压缩和解压缩。你也许需要先安装这些程序。

下一章： usr_24.txt 快速插入

VIM 用户手册 - by Bram Moolenaar
译者：Chimin Yen、tocer

快速插入

输入文本时，Vim 提供你各种各样的方法来减少键击次数和避免打字错误。你可以利用插入模式下的补全功能来重复先前打过的单词。也可以把长词缩写成短词。甚至可以打出你键盘上没有的字符。

24.1 更正

退格键 <BS> 已经在前面提过了。它删除位于光标前一格的字符。而删除键 <Del> 则删除光标下（或者说光标后也可以）的那个字符。
当你把整个词都打错了的时候，用 CTRL-W 来更正：

```
The horse had fallen to the sky
                            CTRL-W
The horse had fallen to the
```

如果你把一行字弄得不可收拾，而要从头来过的话，用 CTRL-U 来删除。这个命令保留了光标之后的文本，也保留了行首的缩进。它只删除了自第一个非空字符至光标位置之间的文本。让光标位于下一行中 "fallen" 的 "f" 上，按 CTRL-U，文本就成了这样：

```
The horse had fallen to the
              CTRL-U
fallen to the
```

当你发现几个词之前有个错误，你需要把光标移到那儿作更正。例如，你打了这样一行：

```
The horse had follen to the ground
```

你要把 "follen" 改成 "fallen"。让光标留在行尾，你输入这个字符串就能更正那错误：

```
                                    <Esc>4blraA

脱离插入模式                        <Esc>
退回 4 个单词                            4b
移到字母 "o" 上                            l
以字母 "a" 替代                             ra
重新开始插入模式                              A
```

另一种更正这个错误的方法:

```
        <C-Left><C-Left><C-Left><C-Left><Right><Del>a<End>

退回 4 个词                  <C-Left><C-Left><C-Left><C-Left>
移到字母 "o" 上                        <Right>
删除字母 "o"                                <Del>
插入字母 "a"                                    a
移到行尾                                         <End>
```

这种方法让你留在插入模式下，利用特殊键来移动光标。这类似于你在一个不分模式的编辑器里所采用的操作方法。它比较容易记忆，但比较费事（你必须把你的手从字母键移动到光标方向键，而且，不看键盘难以按准 <End> 键)。

这些特殊键在书写一个停留在插入模式下的映射时非常有用。额外所需的键盘录入也是值得的。

你在插入模式下可以利用的特殊键有以下这些:

```
<C-Home>        移到文件首
<PageUp>        上卷一屏
<Home>          移到行首
<S-Left>        左移一个单词
<C-Left>        左移一个单词
<S-Right>       右移一个单词
<C-Right>       右移一个单词
<End>           移到行尾
<PageDown>      下卷一屏
<C-End>         移到文件尾
```

还有很多键，参见 ins-special-special 。

24.2 显示匹配

你键入一个右括号) 时，如果能知道它匹配哪个左括号 (会很方便。要让 Vim 做到这点，用下面这个命令:

```
:set showmatch
```

现在你键入比如 "(example)" 这样几个字，当你键入右括号) 时，Vim 就把光标闪到匹配的左括号 (上，在那儿停留半秒钟，然后返回原处。

如果匹配的左括号不存在，Vim 就鸣起响铃。这样你就会想起来你可能在哪儿忘了一个左括号，或键入了太多次的右括号。

Vim 也会为 [] 和 {} 这样的括号显示匹配。你不必等待光标返回原处才键入下一个字符，只要下一个字符一键入，光标就会返回，而插入就会跟以前一样继续。

你可以用 'matchtime' 选项改变 Vim 在匹配括号上停留的时间。例如，要让 Vim 等

待 1.5 秒:

```
:set matchtime=15
```

指定的时间以 1/10 秒为单位。

24.3 补全

Vim 能自动补全插入的单词。你键入一个单词的开头部分，按 CTRL-P，Vim 就会为你猜测余下的部分。
例如，假定你正在建立一个 C 程序，并要键入以下语句:

```
total = ch_array[0] + ch_array[1] + ch_array[2];
```

你先输入下面这部分:

```
total = ch_array[0] + ch_
```

此时，你用命令 CTRL-P 告诉 Vim 来补全这个词。Vim 就会搜索以光标前字符串开头的词。在这个例子中，就是 "ch_"，与词 ch_array 匹配。所以，键入 CTRL-P 就会得到下面的结果:

```
total = ch_array[0] + ch_array
```

再键入几个字符使这个语句变成这样 (结尾是空格):

```
total = ch_array[0] + ch_array[1] +
```

如果你现在键入 CTRL-P，Vim 将再次搜索以补全光标前的词。由于光标前是空格，它找到的是之前的第一个词，即 "ch_array"。再键入 CTRL-P 给你下一个匹配的词，在本例中就是 "total"。第三次 CTRL-P 搜寻更前面的。如果那儿没其它的了，编辑器就会陷入无词可配状态，所以搜索就返回原处，即那个空格。第四次 CTRL-P 导致编辑器周而复始，又找到 "ch_array"。

往下搜索，用 CTRL-N。由于在文件结尾搜索又绕回开头，CTRL-N 和 CTRL-P 将找到相同的匹配，但顺序不同。提示: CTRL-N 意为下一个匹配，而 CTRL-P 意为前一个匹配。
(译者: 英文 Next 意为下一个，Previous 意为前一个)

Vim 编辑器会非常努力的来补全不完整的词。默认情况下，它搜索如下一些地方:

1. 当前文件
2. 其它窗口内的文件
3. 其它载入文件 (隐藏缓冲区)
4. 未载入文件 (非激活缓冲区)
5. 标签文件
6. 被当前文件以 #include 语句包含的所有头文件

选 项

你可以利用 'complete' 选项定制搜索顺序。

还可以使用 'ignorecase' 选项。设定这个选项后，搜寻匹配时大小写的区别就会被忽

略。

一个特殊的补全选项是 'infercase'。它的用处是在寻找忽略大小写的匹配时 ('ignorecase' 必须先被设定)，但仍然采用已键入部分的大小写。这样，如果你键入 "For" 而 Vim 找到了匹配 "fortunately"，所产生的结果将是 "Fortunately"。

补 全 特 定 文 本

如果你知道你要找什么，那么你可以用这些命令来补全某种类型的文本：

CTRL-X CTRL-F	文件名
CTRL-X CTRL-L	整行
CTRL-X CTRL-D	宏定义 (包括包含文件里的)
CTRL-X CTRL-I	当前文件以及所包含的文件
CTRL-X CTRL-K	字典文件内的单词
CTRL-X CTRL-T	同义词词典文件内的单词
CTRL-X CTRL-]	标签
CTRL-X CTRL-V	Vim 命令行

每个命令之后，CTRL-N 可以用来搜索下一个匹配，而 CTRL-P 则用于搜索前一个匹配。
关于每个命令的详细用法，参见： ins-completion 。

补 全 文 件 名

我们以 CTRL-X CTRL-F 为例。这个命令将找寻文件名。它在当前目录里搜索文件，并显示每一个与光标前单词匹配的文件名。
例如，假定你在当前目录里有下面这些文件：

```
main.c  sub_count.c  sub_done.c  sub_exit.c
```

现在进入插入模式并开始键入：

```
The exit code is in the file sub
```

就在这点上，你输入 CTRL-X CTRL-F。现在，Vim 通过查看当前目录里的文件来补全当前词 "sub"。最初的匹配是 sub_count.c。这不是你想要的，所以你按 CTRL-N 以匹配下一个文件。这次匹配的是 sub_done.c。再键入 CTRL-N 给了你 sub_exit.c。结果：

```
The exit code is in the file sub_exit.c
```

如果文件名以 / (Unix) 或 C:\ (MS-Windows) 开头，那么你就能搜索文件系统下所有的文件。例如，键入 "/u" 然后 CTRL-X CTRL-F，这将匹配 "/usr" (这是在 Unix 上)：

```
the file is found in /usr/
```

如果你现在按 CTRL-N，你就又回到 "/u"。接受 "/usr/" 并进入下一层目录，再来一次 CTRL-X CTRL-F：

```
the file is found in /usr/X11R6/
```

当然，匹配结果取决于你的文件系统上有什么文件。匹配结果以字母顺序来排列。

补 全 源 代 码

源代码文件有良好的结构。这使通过某种智能方式补全成为可能。在 Vim 中，这被称为全能补全。在其他编辑器中，它被称为智能补全(intellisense)，但这是一个注册商标。

全能补全的热键是 CTRL-X CTRL-O。显然，O 在这里代表全能 (Omni)，这样方便我们记忆。让我们以编辑 C 程序为例：

```
{
    struct foo *p;
    p->
```

光标在 "p->" 之后。现在键入 CTRL-X CTRL-O。Vim 会给你提供一个可选项的列表，这些可选项为 "struct foo" 所拥有。这和使用 CTRL-P 有很大不同，后者补全任意单词，而我们这里只要求 "struct foo" 的成员。

为使全能补全工作，需要做一些初始化。起码，要保证打开文件类型插件。你的 vimrc 文件应该包含形如：

```
filetype plugin on
```

或：

```
filetype plugin indent on
```

的一行。

对于 C 代码，需要建立标签文件并设置 'tags' 选项。在 ft-c-omni 中有进一步的解释。对于其他文件类型，需要做类似的事情，请查看 compl-omni-filetypes 。补全只对特定文件类型有效。查看 'omnifunc' 选项的值，以便检查补全能否正常工作。

24.4 重复一次插入

如果你按 CTRL-A，编辑器就把你上次在插入模式下输入的文本再输入一次。
比如，假定你有个文件，开头是这样的：

```
"file.h"
/* Main program begins */
```

你在第一行开始处插入 "#include "：

```
#include "file.h"
/* Main program begins */
```

你再用命令 "j^" 往下来到下一行的开始处。现在你开始插入一个新的 "#include" 行。所以你键入：

```
i CTRL-A
```

结果就像下面这样：

```
#include "file.h"
#include /* Main program begins */
```

"#include " 被插入是因为 CTRL-A 会插入上次插入过的文本。现在你键入 "main.h" <Enter> 以结束这一行：

```
#include "file.h"
#include "main.h"
/* Main program begins */
```

CTRL-@ 命令会完成 CTRL-A 的操作后退出插入模式。这是一个快速重复插入一模一样的文本的一个方法。

24.5 从另一行拷贝

CTRL-Y 命令插入光标上方的字符。当你复制前一行文本的时候，这个命令很有用。例如，你有这么一行 C 代码：

```
b_array[i]->s_next = a_array[i]->s_next;
```

现在你需要把这一行再键入一次，并以 "s_prev" 取代 "s_next"。换行以后，按 14 次 CTRL-Y，直到光标位于 "next" 的 "n" 上：

```
b_array[i]->s_next = a_array[i]->s_next;
b_array[i]->s_
```

现在你键入 "prev"：

```
b_array[i]->s_next = a_array[i]->s_next;
b_array[i]->s_prev
```

继续按 CTRL-Y 直到下一个 "next"：

```
b_array[i]->s_next = a_array[i]->s_next;
b_array[i]->s_prev = a_array[i]->s_
```

现在键入 "prev;" 以结束这一行。

CTRL-E 命令操作起来跟 CTRL-Y 一样，只不过它插入光标下方的字符。

24.6 插入一个寄存器内容

命令 CTRL-R {register} 插入寄存器里的内容。它的用处是让你不必键入长词。例如，你要输入下面这些：

```
r = VeryLongFunction(a) + VeryLongFunction(b) + VeryLongFunction(c)
```

这个函数的定义见于另一个文件。编辑那个文件并把光标移到该函数名上，然后把文件名摄入寄存器 v：

```
"vyiw
```

"v 指定寄存器，"yiw" 意思是拷贝一个词，不含空格 (yank-inner-word)。现在编辑那个要插入一行代码的文件，先键入开头几个字符：

```
r =
```

现在用 CTRL-R v 来插入函数名：

```
r = VeryLongFunction
```

你接下来在函数名后面键入其它必要字符，然后再用两次 CTRL-R v。
　你也可以用补全功能来完成同样的工作。但当你有好几个词，其开头几个字符都一样的时候，寄存器就有用多了。

如果寄存器存放着诸如 <BS> 或其它特殊字符，这些字符就被解释成好像它们本来是从键盘键入的。如果你不要这样解释（你确实要在文本中插入 <BS>），那么要命令 CTRL-R CTRL-R {register}。

24.7　缩写

缩写是取代一个长词的短词。例如，"ad" 指代 "advertisement"。Vim 让你键入缩写，然后为你自动扩展。
　用以下命令告诉 Vim，每当你输入 "ad" 就把它扩展成 "advertisement"：

```
:iabbrev ad advertisement
```

现在，当你键入 "ad"，完整的单词 'advertisement" 就被插入文本。键入一个不可能成为单词一部分的字符，例如一个空格，就会触发缩写功能：

```
输入的文本              看到的文本
I saw the a             I saw the a
I saw the ad            I saw the ad
I saw the ad<Space>     I saw the advertisement<Space>
```

当你仅仅键入 "ad" 时，扩展并没发生。它可以被你输入 "add" 这样的词而不被扩展。Vim 只对那些完整的词检查缩写。

多 词 缩 写

为几个词定义一个缩写也是可能的。例如，用下面这个命令，把 "JB" 定义成 "Jack Benny"：

```
:iabbrev JB Jack Benny
```

作为程序员，我使用两个相当不寻常的缩写：

```
:iabbrev #b /*****************************************
:iabbrev #e <Space>*****************************************/
```

它们用于生成大段注释。注释以缩写 #b 开始，划出顶线。接着，我键入注释文字，最后以缩写 #e 划出底线。
　注意 缩写 #e 以一个空格开头。换言之，开头两个字符是空格和星号（*）。通常 Vim 忽略不计位于缩写及其扩展之间的空格。为了避免空格被忽略，我把空格以七个字符表示：<、S、p、a、c、e、>。

备注：

":iabbrev" 有点嫌长。":iab" 作用也一样。缩写命令被缩写了!

更 正 打 字 错 误

我们经常会犯同一个打字错误。例如，把 "the" 打成 "teh"。你可以利用缩写功能来更正这样的错误:

```
:abbreviate teh the
```

你可以加上一系列这样的缩写。每次发现一个常见错误就加一个。

缩 写 列 表

":abbreviate" 命令列出所有缩写:

```
:abbreviate
i  #e             ****************************************/
i  #b            /*****************************************
i  JB            Jack Benny
i  ad            advertisement
!  teh           the
```

第一栏的 "i" 表明插入模式。这些缩写仅仅在插入模式下有作用。其它可能的字符:

```
c       命令行模式                              :cabbrev
!       插入模式和命令行模式                    :abbreviate
```

在命令行模式下使用缩写是不常见的。你主要会在插入模式下用 ":iabbrev" 命令。这样就避免了不必要的扩展，例如，当你键入这样一条命令时，"ad" 就不会被扩展了:

```
:edit ad
```

删 除 缩 写

删除一个缩写，用 ":unabbreviate" 命令。假定你有以下缩写:

```
:abbreviate @f fresh
```

你可以用这个命令删除它:

```
:unabbreviate @f
```

当你键入这个命令的时候，你将注意到 @f 被扩展成 "fresh"。别担心，Vim 明白得很呢(除非当你另有缩写 "fresh"，但那是很偶然的)。

要删除全部缩写:

```
:abclear
```

":unabbreviate" 和 ":abclear" 另有变形，在插入模式下是 "iunabbreviate" 和 ":iabclear"，在命令行模式下是 ":cunabbreviate" 和 "cabclear"。

缩 写 再 映 射

定义缩写时，有一点要 注意 的：扩展产生的字符串不应当被映射成别的什么。例如：

```
:abbreviate @a adder
:imap dd disk-door
```

现在，你键入 @a，你得到 "adisk-doorer"。那不是你要的结果。为了避免这种事发生，用 ":noreabbrev" 命令。它的作用跟 ":abbreviate" 一样，但却避免了扩展产生的字符串被用于映射：

```
:noreabbrev @a adder
```

现在好了，缩写扩展后的结果不可能被映射了。

24.8 插入特殊字符

CTRL-V 命令用来按本义插入下一个字符。换言之，无论该字符多特殊，其特殊含义都被忽略不计。例如：

```
CTRL-V <Esc>
```

插入 <Esc> 字符。而你并未脱离插入模式。(不要在 CTRL-V 后面键入空格，那个空格仅仅为了方便阅读)。

> 备注：
> 在 MS-Windows 环境下，CTRL-V 用来粘贴文本。所以用 CTRL-Q 代替 CTRL-V。另外，在 Unix 环境下，CTRL-Q 在某些终端上不起作用，因其另有特殊意义。

你也可以用命令 CTRL-V {digits} 来插入一个以若干个十进位数字 {digits} 表示的字符。例如，字符编码 127 是字符 <Del> (但并不一定是 <Del> 键！)。要插入 <Del>，键入：

```
CTRL-V 127
```

你可以用这样的方法输入数值不超过 255 的字符。当你键入一个不足两位的数字时，就要添加一个非数字的字符来结束命令。为了回避这个非数字字符的要求，在它前面加一或两个零以满足三位数的要求。

以下命令都插入一个 <Tab> 字符，然后一个点字符：

```
CTRL-V 9.
CTRL-V 09.
CTRL-V 009.
```

输入一个 16 进制数字，在 CTRL-V 后面，用 "x" 开头：

```
CTRL-V x7f
```

这方法也可以用来输入数值不超过 255 (CTRL-V xff) 的字符。你可以用 "o" 开头输入一个以 8 进制数表示的字符，以及另外两种方法，让你输入多至二进制 16 位和 32 位的数字表示的字符 (例如，Unicode 字符)：

```
CTRL-V o123
CTRL-V u1234
CTRL-V U12345678
```

24.9 二合字母

有些字符在键盘上找不到。例如，表示版权的字符（©）要在 Vim 里键入这样的字符，你得用二合字母，即以两个字符来表示一个。例如，要键入 ©， 你就得按三个键：

```
CTRL-K Co
```

你可以用以下命令来查找有哪些二合字母可供利用：

```
:digraphs
```

Vim 将把二合字母表显示在屏幕上。以下就是其中的三行：

```
AC ~_ 159  NS │  160  !I ¡  161  Ct ¢  162  Pd £  163  Cu ¤  164  Ye ¥  165
BB ¦  166  SE §  167  ': ¨  168  Co ©  169  -a ª  170  << «  171  NO ¬  172
--   173  Rg ®  174  'm ¯  175  DG °  176  +- ±  177  2S ²  178  3S ³  179
```

这张表告诉你，比如，你键入 CTRL-K Pd 所得到的二合字母是字符（£）。该字符编码为 163（十进制）。

Pd 是 Pound（英镑）的简写。大多数二合字母让你一看就猜到两个字符生成什么字符。如果你一个个读下来，便不难理解其中的逻辑。

你可以交换两个字符的顺序，只要那样组合不代表另一个二合字母。因此 CTRL-K dP 也没问题。由于 "dP" 不是二合字母，Vim 会转而搜索表示成 "Pd" 的二合字母。

> 备注：
> 二合字母表取决于 Vim 假定你所使用的字符集。务必用 ":digraphs" 命令来查找当前可供利用的二合字符。

你可以定义你自己的二合字母。例如：

```
:digraph a" ä
```

以上命令定义了 CTRL-K a" 插入一个 ä 字符。你也可以用十进值数指定这个字符。下面这个命令定义了同一个二合字母：

```
:digraph a" 228
```

更多关于二合字母信息参见： digraphs

另一种插入特殊字符的方法是利用键盘映射。详情参见： 45.5

24.10 普通模式命令

插入模式提供的命令数量有限。在普通模式下，你可用的命令就多得多了。当你要用一个普通模式命令时，你通常用 <Esc> 键来脱离插入模式，执行这个普通模式命令，然后再用 "i" 或 "a" 命令重新进入插入模式。

有一个快捷的方法。用 CTRL-O {command} 你可以在插入模式下执行任何普通模式命令。例如，把光标后面直至行尾的文本删除：

```
CTRL-O D
```

用这个方法，你只能执行一个普通模式命令。但是，你可以指定一个寄存器或一个计数。
请看下面这个更复杂的命令：

```
CTRL-O "g3dw
```

这个命令把光标后面三个单词一起删除，存进了寄存器 g。

下一章： usr_25.txt 编辑带格式的文本

VIM 用户手册 - by Bram Moolenaar
译者：snoopyxp

编辑带格式的文本

很少有那种一行一个句子的文本。这一章我们将要介绍如何为了适合页面而为句子分行以及进行其它的编排操作。当然，针对编辑单行的段落和表格，Vim 也有一些有用的特性。

下一章： usr_26.txt 重复
前一章： usr_24.txt 快速插入
目录： usr_toc.txt

25.1 断行

Vim 有一系列功能可以使处理文本变得更容易。在默认情况下，Vim 并不会自动换行，换句话说，你得自己敲 <Enter>。这在你编写程序并想自己决定哪里是行尾时会很有用。但当你想建立一个每行至多有 70 个字符宽的文档时，这就显得没那么好了。

如果你设置了 'textwidth' 选项，Vim 就会自动插入换行符。举个例子，假设您想要一个只有 30 个字符宽的非常窄的栏，你需要执行下面的命令：

```
:set textwidth=30
```

现在开始输入正文（加入了标尺）：

```
          1         2         3
123456789012345678901234567890123456
I taught programming for a whi
```

如果接下来你输入 "l"，这将使此行的长度超出 30 个字符的限制。当 Vim 发现这种情况时，它会自动插入换行符。你将得到下面的结果：

```
          1         2         3
123456789012345678901234567890123456
I taught programming for a
whil
```

接着，你可以输入剩下的段落：

```
          1         2         3
123456789012345678901234567890123456
I taught programming for a
while. One time, I was stopped
```

```
by the Fort Worth police,
because my homework wos too
hard. True story.
```

你不用输入换行符，Vim 会自动插入的。

备注：
'wrap' 选项会使 Vim 分行显示文本，但并不会在文件中插入换行符。

重 排 文 本

Vim 编辑器并不是一个字处理器。字处理器在你删除了段落开头的一些东西时会重新调整换行符。(译者注：这意味着后边的文本会向前补进，使换行位置保持不变)。但在 Vim 里并不是这样。因此，当你从第一行删除单词 "programming" 时，你所得到的只是一个短行：

```
         1         2         3
12345678901234567890123456789012345
I taught for a
while. One time, I was stopped
by the Fort Worth police,
because my homework was too
hard. True story.
```

这看起来不大美观。为了保持段落的形状，你要用到 "gq" 操作符。
下面我们首先结合一个可视选择区来使用它。从第一行开始，键入：

```
v4jgq
```

"v" 用来进入可视模式，"4j" 用来移动到段落结尾，接下来是 "gq" 操作符。下面是得到的结果：

```
         1         2         3
12345678901234567890123456789012345
I taught for a while. One
time, I was stopped by the
Fort Worth police, because my
homework was too hard. True
story.
```

备注：有一种方法可以自动地排版特定文本类型的布局，参见 auto-format 。

因为 "gq" 是一个操作符，所以你可以通过下面三种方法之一来选择要操作的文本范围：可视模式，一个移动操作，或是一个文本对象。
上面的例子也可以用 "gq4j" 来完成。这减少了击键次数，但你得知道行数。一个更有用的移动命令是 "}"。使用这个命令可以移动到一个段落的结尾处。因此，"gq}" 将对从当前光标处至当前段段尾所包括的文本进行排版操作。
可以和 "gq" 一起使用的一个很有用的文本对象是段落。试一下这个：

```
gqap
```

"ap" 意味着 "a-paragraph" (译者注：一个段落)。这将对一个段落 (用空行隔开) 的文本进行排版操作。也包括在光标之前的部分。

如果你所有的段落都已经用空行分隔好了，你可以键入以下命令来对整个文件进行格式化操作：

```
gggqG
```

"gg" 用来移动到第一行，"gqG" 用来对文本进行排版操作直到最后一行。

警告：如果你的段落没有被正确的分隔开，它们将被连接在一起。一个常见的错误是文件内包含一些只有空格或制表符的行。这些是空白行，但不是空行。

Vim 并不是只能对纯文本进行排版操作。请参考 fo-table 一节获取更详细的信息。参考 'joinspaces' 选项以了解如何改变在句号之后加入的空格数。

也可以调用外部程序来做排版。这在 Vim 自带的排版功能无法满足你的文本的需要时很有用。参阅 'formatprg' 选项。

25.2 对齐文本

要使一定范围包括的行居中，用以下命令：

```
:{range}center [width]
```

{range} 即是那些通常命令所能指定的范围。 [width] 是一个可选项，用来确定要居中的行的宽度。如果没有指定 [width]，则默认使用 'textwidth' 的值。(如果 'textwidth' 是 0，则默认为 80。)

例如：

```
:1,5center 40
```

将得到下面的结果：

```
      I taught for a while. One
      time, I was stopped by the
    Fort Worth police, because my
     homework was too hard. True
                story.
```

右 对 齐

类似地，":right" 命令将使文本右对齐：

```
:1,5right 37
```

将得到这种结果：

```
           I taught for a while. One
          time, I was stopped by the
       Fort Worth police, because my
         homework was too hard. True
                               story.
```

左 对 齐

我们使用这个命令：

```
:{range}left [margin]
```

和 ":center" ":right" 不同的是，":left" 的参数并不是行的宽度，而是左边的页边距。如果这个参数被省略了，文本将被紧靠在屏幕的左边缘（用一个零页边距参数将得到一样的效果）。如果页边距是 5，文本将缩进五个空格。举个例子，使用以下命令：

```
:1left 5
:2,5left
```

结果会是下面这样：

```
     I taught for a while. One
time, I was stopped by the
Fort Worth police, because my
homework was too hard. True
story.
```

两 端 对 齐 **justify :Justify Justify() package-justify**

Vim 并没有提供内建的方法来使文本两端对齐。不过，我们可以通过使用一个灵巧的宏来做这项工作。要使用这个包，执行下列命令：>vim

```
:packadd justify
```

或在 .vimrc 里放上此行： >vim

```
:packadd! justify
```

这个 Vim 脚本文件定义了一个新的可视命令 "_j"。要使一块文本两端对齐，只需在可视模式中选择该文本块，然后执行 "_j"。

请查阅该文件以获得更详细的解释说明。在文件名 $VIMRUNTIME/pack/dist/opt/justify/plugin/justify.vim 上键入命令 "gf"，就可以打开该文件。

作为另一个选择，你还可以使用一个外部程序来对文本进行过滤。例如：

```
:%!fmt
```

25.3 缩进和制表符

缩进可以被用来突显特定的文本。举个例子，在本手册的示例文本用八个空格或一个制表符来缩进。通常，你可以在每行的开头键入一个制表符来输入下面内容。用这个文本：

```
the first line
the second line
```

这些文本是这样输入的，先输入一个制表符，再输入一些文本，然后键入 <Enter>，接着又是一个制表符，然后输入另一些文本。

置位 'autoindent' 选项可以自动插入缩进：

```
:set autoindent
```

当开始一个新行时，新行会采用和上一行相同的缩进。在上面的例子中，我们再也不需要在 <Enter> 后面输入制表符了。

增 加 缩 进

要增加一行中的缩进量，可以使用 ">" 操作符。一个经常使用的操作是 ">>"，这将为当前行增加缩进。

增加的缩进量是使用 'shiftwidth' 来指定的。默认的值是 8。举例来说，要使 ">>" 插入四个空格宽度的缩进，键入：

```
:set shiftwidth=4
```

当你在示例文本中的第二行上使用时，你会得到下面的结果：

```
the first line
    the second line
```

"4>>" 将为四行增加缩进。

制 表 位

如果你想使缩进量为 4 的倍数，你需把 'shiftwidth' 设置为 4。但是当你敲下 <Tab> 键时，你仍然会得到八个空格宽度的缩进。要改变这种情况，请设置 'softtabstop' 选项。

```
:set softtabstop=4
```

这将使 <Tab> 键插入四个空格宽度值的缩进量。如果已经存在四个空格，就加上一个 <Tab> 字符（在文件中省去了七个字符）。(如果你只想要空格而不想加上 tab 字符，请置位 'expandtab' 选项。)

备注：
你可以把 'tabstop' 选项设置为 4。尽管如此，如果你在另一个时间再次编辑这个文件时 'tabstop' 的默认值是 8，文件缩进看起来会不对。在其它程序中或者在打印的时候，缩进也将是错的。因此，建议把 'tabstop' 的值始终保持为 8。这是所有地方的标准值。

更 换 制 表 位

当你在 Vim 中编辑一个 tabstop 为 3 的文件时，文件会看起来很难看，因为在 Vim 中 tabstop 的正常值为 8。你可以通过把 'tabstop' 的值设置为 3 来更正它。但是你每一次你编辑这个文件都得做这个更改。

Vim 可以更换你文件使用的制表位。首先，设置 'tabstop' 的值使缩进看起来美观，然后使用 ":retab" 命令：

```
:set tabstop=3
:retab 8
```

":retab" 命令将把 'tabstop' 的值改为 8，同时仍保持文件看起来不变。它把空白区间用制表和空格重新构造。现在你可以写入这个文件。下次你再编辑它的时候缩进将是正确的，你不需要改变任何选项。

警告：当你对一个程序文件使用 ":retab" 命令时，它可能会改变一个字符串常量中的空白。因此，要养成在程序中使用 "\t" 而不是输入一个制表符的好习惯。

25.4 对长行的处理

有时你会编辑一个比窗口列数宽的文件。当发生这种情况时，Vim 将把文件行回绕显示以便适应屏幕显示。

如果你将 'wrap' 选项设置为关闭，文件中的每一行都将在屏幕上仅作为一行显示。这时行尾会超出屏幕右端从而无法看到。

当你移动光标到视野之外的字符时，Vim 将滚动文本来显示它。这就好像在文本的水平方向移动视窗一样。

默认情况下，Vim 并不在 GUI 中显示水平滚动条。如果你想启用它，使用下面的命令：

```
:set guioptions+=b
```

一个水平滚动条将出现在 Vim 窗口的底部。

如果你没有滚动条或者你不想使用它，可以用下面这些命令来滚动文本。光标将停留在同样的地方，但在必要时会回移到可见文本区。

```
zh              向右滚动
4zh             向右滚动四个字符
zH              向右滚动半个窗口宽度
ze              向右滚动使光标处于行尾
zl              向左滚动
4zl             向左滚动四个字符
zL              向左滚动半个窗口宽度
zs              向左滚动使光标处于行首
```

让我们试着用一行文本来演示一下。光标停留在 "which" 的 "w" 处。那行上方的 "当前窗口" 标示当前可见的文本。下方的 "窗口" 指示了执行了左边的命令后可见的文本区域。

```
                        |<--    当前窗口    -->|
        some long text, part of which is visible in the window
ze        |<--      窗口      -->|
zH         |<--      窗口      -->|
4zh              |<--      窗口      -->|
zh                  |<--      窗口      -->|
zl                    |<--      窗口      -->|
4zl                      |<--      窗口      -->|
zL                             |<--      窗口      -->|
zs                            |<--      窗口      -->|
```

在关闭折行情况下移动

当 'wrap' 选项被关闭，文本将在水平方向卷动。你可以通过以下命令来使光标移动到你可以看到的一个字符处。这样超出窗口左右两端的文本将被忽略。这些命令不会使文本卷动：

```
g0              移动到当前行的第一个可见字符
```

```
g^              移动到当前行的第一个非空白的可见字符
gm              移动到屏幕行的中点
gM              移动到当前行的文本中点
g$              移动到当前行的最后一个可见字符

        |<--       窗口       -->|
some long    text, part of which is visible in one line
         g0  g^    gm      gM g$
```

断 词 **edit-no-break**

当你为另一个程序准备文本时，你或许需要使段落没有一处换行。使用 'nowrap' 选项的一个弊端是你看不见你正在处理的整个句子。当 'wrap' 选项开启时，会从单词中间断开，从而难以阅读。

编辑此类段落时，一个好的解决方法是设置 'linebreak' 选项。这样，Vim 将会在一个适当的地方回绕行显示，同时仍保持文件中的文本不变。

没有设置 'linebreak' 选项时的文本看起来可能是这样：

```
+---------------------------------+
|letter generation program for a b|
|ank.  They wanted to send out a s|
|pecial, personalized letter to th|
|eir richest 1000 customers.  Unfo|
|rtunately for the programmer, he |
+---------------------------------+
```

使用如下命令之后：

```
:set linebreak
```

看起来会是这样：

```
+---------------------------------+
|letter generation program for a  |
|bank.  They wanted to send out a |
|special, personalized letter to  |
|their richest 1000 customers.    |
|Unfortunately for the programmer,|
+---------------------------------+
```

相关选项：
'breakat' 指定了可以用来作为插入换行地点的字符。
'showbreak' 指定了一个用于显示在回绕行显示行的行首的字符串。
设置 'textwidth' 的值为零来避免一个段落被拆分。

在 可 见 行 间 移 动

使用 "j" 和 "k" 命令可以移动到下一行和上一行。当作用于一个长行时，这通常意味着要一次移动许多屏幕行。

使用 "gj" 和 "gk" 命令可以只移动一个屏幕行。当一行没有回绕时，它们和 "j" 和 "k" 命令所起的作用一样。当一行回绕时，它们将在屏幕上显示的上一行和下一行的一个字符之间移动。

下面这些移动命令的键映射也许对你有用：

```
:map <Up> gk
:map <Down> gj
```

变 段 为 行 **edit-paragraph-join**

如果你想把文本导入类似 MS-Word 的程序中，每个段落就要变成一个单行。如果你的段落是由空行分隔开的，下面这个命令就可以要把一个段转化为一个单行：

```
:g/./,/^$/join
```

这看起来挺复杂。让我们把它分解开：

```
:g/./          一个 ":global" 命令，用来搜索至少含一个字符的所有行。
     ,/^$/     一个范围，从当前行开始（非空行）到一个空行。
          join "join" 命令把范围内的行连接成一行。
```

从下面这段含有八行，且在第三十列换行的文本开始：

```
+----------------------------------+
|A letter generation program       |
|for a bank.  They wanted to       |
|send out a special,               |
|personalized letter.              |
|                                  |
|To their richest 1000             |
|customers.  Unfortunately for     |
|the programmer,                   |
+----------------------------------+
```

你将得到下面两行：

```
+----------------------------------+
|A letter generation program for a |
|bank.  They wanted to send out a s|
|pecial, personalized letter.      |
|To their richest 1000 customers.  |
|Unfortunately for the programmer, |
+----------------------------------+
```

注意 当分隔段落的行是含有空格和/或制表符的空白行而不是空行时，这行命令将不起作用。下列的命令对于空白行仍起作用：

```
:g/\S/,/^\s*$/join
```

要使最后一段也被连接，这行命令需要文件的结尾仍有一个空白行或空行。

25.5 编辑表格

设想你正在编辑一个含有四列的表格：

```
nice table       test 1       test 2      test 3
```

```
input A           0.534
input B           0.913
```

你需要在第三列输入数字。你可以先移动到第二行，键入 "A"，键入一大堆空格然后输入你要输入的文本。

对于这种类型的编辑操作有一个特殊的选项：

```
set virtualedit=all
```

现在你可以把光标移动到没有任何文本的位置。这叫做 "虚拟空间"。通过这种方法，编辑表格变得容易很多。

通过查找最后一列的标题来移动光标：

```
/test 3
```

现在按下 "j"，光标就到了你要输入对应 "input A" 的值的地方了。输入 "0.693"，结果是：

```
nice table        test 1     test 2      test 3
input A           0.534                  0.693
input B           0.913
```

Vim 已经自动为你填充了新文本前面的间隙。现在，要到此列的下一个域，键入 "Bj"。"B" 用来反向移动到空白间隔的单词的开始处。然后，用 "j" 移动到可以输入下一域的地方。

备注：
你可以把光标移动到窗口的任何地方，也可以超出行尾。但是，Vim 在你未在该位置插入字符之前并不会插入空格。

拷 贝 一 列

你想增加一列，它是第三列的一个拷贝，并且想放在 "test 1" 列的前面。做以下七步：

1. 移动光标到这一列的左上角，例如用 "/test 3"。
2. 按 CTRL-V 来开启可视列块模式。
3. 用 "2j" 使光标向下移动两行。你现在进入了 "虚拟空间": "input B" 行中对应对应列 "test 3" 的地方。
4. 把光标向右移动来选择整列，外加那些你想要的列之间的空格。用 "9l" 就行。
5. 用 "y" 来拷贝矩形选择区的内容。
6. 把光标移动到我们需要插入新列的地方 "test 1"。
7. 按下 "P"。

结果应该是：

```
nice table        test 3     test 1     test 2      test 3
input A           0.693      0.534                  0.693
input B                      0.913
```

需要注意的是整个 "test 1" 列被向右移动了，同时也包括 "test 3" 列中没有文本的行。

要返回到光标的非虚拟移动模式，用：

```
:set virtualedit=
```

虚 拟 替 换 模 式

使用 'virtualedit' 的弊端是你会 "感觉" 不太一样。当你移动光标的时候，你不能分辨超出行尾的制表符或空格。另一种可行的方法是：虚拟替换模式。
　设想在表格中有一个包含了制表符和其它字符的行。在第一个制表符上使用 "rx"：

```
inp     0.693   0.534   0.693

          |
    rx    |
          V

inpx0.693   0.534       0.693
```

版面被弄乱了。要避免这种情况，使用 "gr" 命令：

```
inp     0.693   0.534   0.693

          |
    grx   |
          V

inpx    0.693   0.534   0.693
```

这里的情况是 "gr" 命令确保了新字符占据了正确的屏幕空间。额外需要的空格或制表符被插入到间隙中。然而，真正发生的是一个制表符被一个 "x" 代替，然后空白字符被加上来使文本保持它的位置。在这个例子里是插入了一个制表符。
　当你需要替换多于一个字符时，使用 "R" 命令来进入替换模式（参看 04.9 ）。这样一来，原来的版面乱套了，而且替换了不该换的字符。

```
inp     0       0.534   0.693

          |
   R0.786 |
          V

inp     0.78634 0.693
```

"gR" 使用虚拟替换模式。这保全了版面布局：

```
inp     0       0.534   0.693

          |
  gR0.786 |
          V

inp     0.786   0.534   0.693
```

下一章： usr_26.txt 重复

版权: 参见 manual-copyright vim:tw=78:ts=8:noet:ft=help:norl:

VIM 用户手册 - by Bram Moolenaar
译者：Chimin Yen

重复

编辑工作往往不是杂乱无章的。同一个修改动作常常会被重复多次。在本章中，我们将解释几种有用的重复修改动作的方法。

下一章： usr_27.txt 查找命令及模式
前一章： usr_25.txt 编辑带格式的文本
目录： usr_toc.txt

26.1 可视模式下的重复

可视模式在反复修改一段文本方面十分方便。你可以看见高亮的文本，从而确定是否改对了地方。但选择文本也得打几个字。"gv" 命令再次选择上次选过的文本，让你对同一段文本再作别的修改。

假定你有几行文本，你要把其中的 "2001" 改成 "2002"，以及 "2000" 改成 "2001":

```
The financial results for 2001 are better
than for 2000.  The income increased by 50%,
even though 2001 had more rain than 2000.
                2000            2002
income          45,403          66,234
```

首先把 "2001" 改成 "2002"。在可视模式下选择这几行文本，并执行:

```
:s/2001/2002/g
```

现在用 "gv" 命令再次选择同一文本。光标在哪儿都没关系。然后用 ":s/2000/2001/g" 命令作第二项修改。

很明显，你可以多次重复这些动作。

26.2 加与减

当你反复地把一个数改成另一个时，你常遇到定值的偏移。在上例中，每个年份都加了一。与其为每个要修改的年份都键入一个替换命令，不如利用 CTRL-A 命令。

在上面这段文本中搜索年份:

```
/19[0-9][0-9]\|20[0-9][0-9]
```

现在按 CTRL-A。找到的年份增加了一：

```
The financial results for 2002 are better
than for 2000.  The income increased by 50%,
even though 2001 had more rain than 2000.
                2000            2001
income          45,403          66,234
```

用 "n" 命令找到下一个年份，并按 "." 重复 CTRL-A 命令（键入 "." 会快一点儿）。为所有找到的年份重复 "n" 和 "."。

提示：设定 'hlsearch' 选项以突显那些你要修改的数字，然后你可以提前观察并修改得快些。

你可以在 CTRL-A 之前附加数字来增加一个大于一的数。假定你有这么个列表：

```
1. item four
2. item five
3. item six
```

把光标移到 "1." 上并键入：

```
3 CTRL-A
```

那个 "1." 就变成了 "4."。同样，你可以用 "." 命令对其它数字重复这项修改。

再看一个例子：

```
006     foo bar
007     foo bar
```

在这些数字上执行 CTRL-A 产生了以下结果：

```
007     foo bar
010     foo bar
```

7 加 1 等于 10？这是因为 Vim 根据首位出现的 "0" 而误以为 "007" 是个八进位数字。这种表示法常用于 C 程序。如果你不要首位为 "0" 的数字被处理成八进位数字，请用这个命令：

```
:set nrformats-=octal
```

CTRL-X 命令用于减数字，用法与加数字类似。

26.3 改动多个文件

假定你有个变量名为 "x_cnt" 而你要把他改为 "x_counter"。这个变量在多个 C 文件都被用到了。你需要在所有文件中作此改动。你得这么做。

把所有相关文件放进参数列表：

```
:args *.c
```

这个命令会找到所有的 C 文件并编辑其中的第一个。现在你可以对所有这些文件执行替代命令：

```
:argdo %s/\<x_cnt\>/x_counter/ge | update
```

命令 ":argdo" 把另一个命令当作其参数。而后者将对参数列表内所有的文件执行。
作为参数的替代命令 "%s" 作用于所有文本行。它用 "\<x_cnt\>" 来查找 "x_cnt"。"\<" 和 "\>" 用来指定仅匹配那些完整的词，而不是 "px_cnt" 或 "x_cnt2"。
替代命令的标志位中包含 "g"，用以置换同一行文本内出现的所有的匹配 "x_cnt"。标志位 "e" 用于避免因文件中找不到 "x_cnt" 而出现错误信息。否则 ":argdo" 命令就会在遇到第一个找不到 "x_cnt" 的文件时中断。
字符 "|" 分隔两条命令。后面的 "update" 命令将那些有改动的文件存盘。如果没有 "x_cnt" 被改成 "x_counter"，这个命令什么也不做。

还有一个 ":windo" 命令，用于在所有窗口内执行其参数所规定的命令。以及 ":bufdo" 命令，对所有缓冲区执行其参数所规定的命令。使用中要小心，因为你在缓冲区列表中的文件数量可能超过你能想像的。请用 ":buffers" 命令（或 ":ls"）来检查缓冲区列表。

26.4 在外壳脚本里使用 Vim

假定你要在很多文件内把字符串 "-person-" 改成 "Jones"，然后把它们打印出来。你该怎么做？一种方法是键入许许多多命令。另一种是写个外壳脚本来完成这件工作。
作为一个可视的交互式的编辑器，Vim 在执行普通模式命令时表现得极为出色。然而在批量处理时，普通模式命令无法产生简洁的带有注释的命令文件；在此，你该转而利用 Ex 模式。该模式为你提供一种友好的命令行界面，方便你把命令写进一个批处理文件。("Ex 命令" 无非是命令行 (:) 命令的另一个名称。)
以下就是你所需要的 Ex 模式命令：

```
%s/-person-/Jones/g
write tempfile
quit
```

你把这些命令放进文件 "change.vim" 里。现在就用这个外壳脚本在批量模式下运行编辑器：

```
for file in *.txt; do
  vim -e -s $file < change.vim
  lpr -r tempfile
done
```

循环 for-done 是一个外壳结构，用来重复执行循环结构内的两行命令，而变量 $file 则在每次循环时被设成不同的文件名。
脚本第二行的作用是对文件 $file 运行 Vim 且在 Ex 模式下 (参数 -e)，并从文件 "change.vim" 读取命令。参数 -s 告诉 Vim 运行在安静模式下。换言之，不要持续不断的发出 :prompt，或针对那个问题发出的任何别的提示。
命令 "lpr -r tempfile" 打印执行中产生的文件 "tempfile" 并把它删除 (是参数 -r 的作用)。

从标准输入读取

Vim 能从标准输入读取文本。由于通常从那里读取的是命令，你得告诉 Vim 你读的是文本。这需要在通常是文件名的地方传送一个参数 "-"。例如：

```
ls | vim -
```

这个命令让你编辑 "ls" 命令的输出结果，而不必先把那些输出文本存入一个文件。
　如果你从标准输入读取文本，那么你可以用参数 "-S" 来读取脚本：

```
producer | vim -S change.vim -
```

普 通 模 式 脚 本

如果你真的要在脚本内利用普通模式命令，你可以这样来用：

```
vim -s script file.txt ..。
```

注意：
当 "-s" 不与 "-e" 一起用时，它的意思是不同的。此时，它的意思是把文件 "script" 里的命令当作普通模式命令来执行。而与 "-e" 一起用时，它的意思是保持安静，并不会把下一个参数视为文件名。

文件 "script" 里的命令就像你键入它们那样得到执行。别忘了换行符被解释成按下回车键 <Enter>。在普通模式下该键把光标移到下一行。
　要创建这么一个脚本你可以编辑这个脚本文件，键入那些命令。你得想象每个命令会产生什么样的结果。这可不那么容易。另一种方法是在你手动执行那些命令时把它们记录下来。你可以采用下面的方法：

```
vim -w script file.txt ..。
```

所有键入的字符都将被写进文件 "script"。如果你犯了个小错误，不妨继续输入。但要记得事后更正一下这个脚本文件。
　参数 "-w" 会将新键入的命令附加在一个已存在的脚本文件末尾。这在你需要一点儿一点儿记录该脚本时是很不错的。但当你要从零开始重新记录你的脚本时，你则需要用 "-W" 参数。该参数重写任何已存在的文件。

下一章：　usr_27.txt　　查找命令和模式

VIM 用户手册 - by Bram Moolenaar
译者：Chimin Yen

查找命令及模式

在第三章里，我们曾经提到过几个简单的查找模式 03.9 。Vim 能够胜任复杂得多的查找。本章将解释那些最常用到的模式。详细的说明可以查阅： pattern

下一章：　usr_28.txt　折叠
前一章：　usr_26.txt　重复
目录：　usr_toc.txt

27.1　忽略大小写

默认情况下，Vim 的查找是大小写敏感的。因此，"include"，"INCLUDE"，和 "Include" 为三个各不相同的词，而一次查找将仅仅匹配其中的一个。
现在设定 'ignorecase' 选项：

```
:set ignorecase
```

再查找 "include"，现在它将匹配 "Include"，"INCLUDE" 和 "InClUDe"。(设定 'hlsearch' 选项可以快速查看哪儿有模式的匹配。)
你可以这样撤销这个选项：

```
:set noignorecase
```

让我们保留这个设定，并查找 "INCLUDE"。它匹配的文本就跟查找 "include" 时一模一样。现在我们设定 'smartcase' 选项：

```
:set ignorecase smartcase
```

如果你采用的模式里至少有一个大写字母，查找就成了大小写敏感的。可以这样理解，你不必键入大写字母，也能查找到含有大写字母的词，所以，你若键入大写字母，必然是因为你要大小写敏感的匹配。聪明!
设定了这两个选项，你会找到以下匹配：

模式	匹配
word	word、Word、WORD、WoRd 等。

```
Word                    Word
WORD                    WORD
WoRd                    WoRd
```

单 个 模 式 里 的 大 小 写

如果你仅想忽略一个特定的模式里的大小写，那么，在模式前添加 "\c" 字符串就行了。添加 "\C" 将使该模式的匹配大小写敏感。这排除了 'ignorecase' 和 'smartcase' 选项的影响，当 "\c" 或 "\C" 起作用时，它们设成什么值无关紧要。

模式	匹配
\Cword	word
\CWord	Word
\cword	word，Word，WORD，WoRd，等。
\cWord	word，Word，WORD，WoRd，等。

采用 "\c" 和 "\C" 的好处在于它粘附在所用的模式上。因此，重复查找历史记录里的某个模式会产生同样的结果。'ignorecase' 或 'smartcase' 是否更改不影响结果。

备注：
在查找模式中使用以 "\" 开头的项，其效果取决于 'magic' 选项。在本章中，我们将假定 'magic' 为真。这也是标准和建议的设定。如果你把它改了，那么，许多查找模式就会突然变得无效了。

备注：
如果你查了好久，超过了你预计的时间，你可以中断查找，在 Unix 上用 CTRL-C，而在 MS-Windows 上则用 CTRL-Break。

27.2 在文件尾折返

在默认情况下，正向查找从当前光标位置开始，查找特定的字符串。然后它就遇到了文件尾。如果那个时候还没找到那个字符串，它就从头来过，从文件开头一直查到光标处。

注意，当你不断以 "n" 命令查找下一个匹配时，你最终回到第一个匹配。如果你不，你将永远找下去！为了提醒你，Vim 显示如下信息：

```
search hit BOTTOM, continuing at TOP
```

如果你使用 "?" 命令按相反方向查找，你得到的信息是这样的：

```
search hit TOP, continuing at BOTTOM
```

可是，你还是不知道你何时回到了第一个匹配。一种办法是设定 'ruler' 选项：

```
:set ruler
```

Vim 将把光标位置显示在窗口的右下角（如果使用了状态条的话，会显示在那里）。看起来像这样：

```
101,29        84%
```

第一个数字是光标所在的行号。在你开始查找的时候记住行号，那样你就能检查是否越过

了该位置。

无 折 返 查 找

要取消查找折返功能，请用以下命令：

```
:set nowrapscan
```

现在当查找遇到文件末尾，一个出错信息就会显示出来：

```
E385: search hit BOTTOM without match for: forever
```

这样，你只要用 "gg" 命令回到文件开头，并一直查到你看到以上信息，你就能找到所有的匹配。

如果你用 "?" 从相反方向查找，你将得到：

```
E384: search hit TOP without match for: forever
```

27.3 偏移

在默认情况下，查找命令让光标停留在匹配的模式的开始。你可以指定一个偏移，告诉 Vim 将光标停留在别的位置上。在正向查找命令 "/" 中指定偏移，就是在模式后面附加一个斜线符（/）以及偏移值：

```
/默认/2
```

这个命令查找模式 "默认"。找到后使光标越过匹配的模式而下移两行，并停留在该行的行首。把这个命令用于以上段落中，Vim 在第一行找到词 "默认"。接着光标再往下移两行，落在 "一个" 的 "一" 上。

如果该偏移为一简单数字，那么光标就会被放置在距离匹配那么多行的那一行的行首。该偏移值可为正数或负数。如果它是正数，光标会向下移该数表示的行；若为负数，则往回退该数表示的行。

字 符 偏 移

偏移符 "e" 表示一个偏移从匹配末尾算起。它把光标移到匹配的最后一个字符上。命令：

```
/const/e
```

把光标放到单词 "const" 的 "t" 上。

加一个数字，光标就从该位置再前移该数字指定的那么多个字符。下面这个命令会将光标移到匹配后面第一个字符：

```
/const/e+1
```

一个正数使光标右移，负数使其左移。例如：

```
/const/e-1
```

会把光标移到单词 "const" 的 "s" 字符上。

如果偏移以 "b" 开头，那么光标就移到匹配模式的首位。因为不用 "b" 光标也一样会被移到首位，所以单独使用时没什么意义。在将它与一个加上或减去的数字合起来时，就很有用了。光标就会前移或后移那么多个字符。例如：

```
/const/b+2
```

会把光标移到匹配的首位，再往右移两个字符。因而落在字符 "n" 上。

重 复

当你重复前一次使用过的查找模式，只是偏移不同时，你可以把模式省略了：

```
/that
//e
```

等于：

```
/that/e
```

再以同样的偏移重复查找：

```
/
```

命令 "n" 具有同样的作用。要取消一个以前用过的偏移可以用：

```
//
```

反 向 查 找

命令 "?" 以相同的方式使用偏移，但你必须以 "?" 来分隔模式和偏移，而非 "/"：

```
?const?e-2
```

偏移符 "b" 和 "e" 的用途是一样的。它们并不因为使用了 "?" 而改变方向。

起 始 位 置

查找时，通常从光标位置开始。当你规定的是一个行偏移，这可能造成麻烦。例如：

```
/const/-2
```

这个命令找到下一个单词 "const"，然后上移两行。如果你用命令 "n" 再找，Vim 就从当前位置开始，找到同一个 "const" 匹配。然后再一次在偏移的作用下，回到开始的地方。你给套住了！
　还有比这更糟糕的：假定下一行另有一个 "const" 匹配。那么，重复正向查找就会找到这个匹配，并上移两行。这样你实际上把光标往回移了！

当你规定的是一个字符偏移，Vim 将为其作调整。因此，查找会向前或向后跳过几个字符再开始，以便同一个匹配不至于再出现。

27.4 匹配重复性模式

星号项 "*" 规定在它前面的项可以重复任意次。因此:

```
/a*
```

匹配 "a", "aa", "aaa", 等等。但也匹配 "" (空字串), 因为零次也包含在内。
星号 "*" 仅仅应用于那个紧邻在它前面的项。因此 "ab*" 匹配 "a"、"ab"、"abb"、"abbb" 等等。如要多次重复整个字符串, 那么该字符串必须被组成一个项。组成一项的方法就是在它前面加 "\(", 后面加 "\)"。因此这个命令:

```
/\(ab\)*
```

匹配: "ab", "abab", "ababab", 等等。而且也匹配 ""。

要避免匹配空字串, 使用 "\+"。这表示前面一项可以被匹配一次或多次。

```
/ab\+
```

匹配 "ab"、"abb"、"abbb" 等等。它不匹配后面没有跟随 "b" 的 "a"。

要匹配一个可选项, 用 "\="。 例如:

```
/folders\=
```

匹配 "folder" 和 "folders"。

指 定 重 复 次 数

要匹配某一项的特定次数重复, 使用 "\{n,m}" 这样的形式。其中 "n" 和 "m" 都是数字。在它前面的那个项将被重复 "n" 到 "m" 次 (inclusive 包含 "n" 和 "m")。例如 :

```
/ab\{3,5}
```

匹配 "abbb"、"abbbb" 以及 "abbbbb"。
当 "n" 省略时, 被默认为零。当 "m" 省略时, 被默认为无限大。当 ",m" 省略时, 就表示重复正好 "n" 次。例如:

模式	匹配次数
\{,4}	0, 1, 2, 3 或 4
\{3,}	3, 4, 5 等等
\{0,1}	0 或 1, 同 \=
\{0,}	0 或更多, 同 *
\{1,}	1 或更多, 同 \+
\{3}	3

匹配尽可能少的字符

迄今为止，我们所讨论过的都试图匹配尽可能多的字符。若要匹配尽可能少的字符，请用 "\{-n,m}"。它的用法跟 "\{n,m}" 一样，唯一的区别在于，它采用尽可能少的字符。
例如，以下命令：

```
/ab\{-1,3}
```

将匹配 "abbb" 中的 "ab"。实际上，因为没理由匹配更多，所以它永远不会匹配超过一个 b。它需要其它的来强制它超过其下限规定次数，而匹配更多的重复。
这些同样的规则也适用于省略 "n" 和 "m" 的情形。甚至可以把两个都省略，只剩 "\{-}"。这个项匹配其前项的重复，重复次数尽可能少，可以等于或大于零。这个项如单独使用，则总是匹配前项的零次重复。当它跟与其它的模式合起来时，用处就大了。例如：

```
/a.\{-}b
```

这个命令匹配 "axbxb" 中的 "axb"。如果采用了下面这个模式：

```
/a.*b
```

由于 ".*" 匹配尽可能多的字符，整个 "axbxb" 都会被匹配。

27.5 多择一

在一个查找模式中，"或" 运算符是 "\|"。例如：

```
/foo\|bar
```

这个命令匹配了 "foo" 或 "bar"。更多的抉择可以连在后面：

```
/one\|two\|three
```

匹配 "one"、"two" 或 "three"。
如要匹配其多次重复，那么整个抉择结构须置于 "\(" 和 "\)" 之间：

```
/\(foo\|bar\)\+
```

这个命令匹配 "foo"、"foobar"、"foofoo"、"barfoobar" 等等。
再举个例子：

```
/end\(if\|while\|for\)
```

这个命令匹配 "endif"、"endwhile" 和 "endfor"。

一个与此相关的项是 "\&"。它要求两个抉择都与同一位置的文本相符。而最终匹配的则是最后面的那个抉择。例如：

```
/forever\&...
```

这个命令匹配 "forever" 中的 "for" (译者：因为第二抉择要求三个字符)。它将不匹配，比如说 "fortuin" (译者：不符合第一抉择)。

27.6 字符范围

你可以用 "/a\ b\ c" 来匹配 "a"，"b" 或 "c"。当你需要匹配自 "a" 至 "z" 所有的字母时，以这样的方式表达就嫌长了。这里有个比较简短的表达方式：

```
/[a-z]
```

方括号结构 [] 匹配单个字符。你在括号内指定哪些字符可以匹配。你可以把一系列字符包含在内，像这样：

```
/[0123456789abcdef]
```

这个命令将匹配其中的任何一个字符。你可以为一系列连续字符规定一个字符范围。"0-3" 表示 "0123"。"w-z" 表示 "wxyz"。因此，上面那个命令可以缩短为：

```
/[0-9a-f]
```

若要匹配字符 "-" 本身，就得把它放在字符范围的第一或最后的位置上。Vim 会识别下面这些特殊字符，以便在 [] 字符范围里较为方便地使用它们（它们实际上可被用于任何查找模式的任何地方）：

```
\e      <Esc>
\t      <Tab>
\r      <CR>
\b      <BS>
```

还有若干特殊场合用得上 [] 字符范围，参阅 /[] 以了解全部用法。

范 围 求 反

为了避免匹配到一个特定的字符，在字符范围首位使用 "^"。这样方括号项 [] 就会匹配任何括号内不包括的字符。例如：

```
/"[^"]*"

 "       双引号
  [^"]   双引号以外的任何字符
     *   尽可能多个
      "  又一个双引号
```

这个命令匹配 "foo" 和 "3!x"，包含双引号在内。

预 定 义 范 围

有些字符范围使用得很频繁。Vim 为这些字符范围提供了一些快捷方式。例如：

```
/\a
```

这个命令找寻字母字符。这相当于使用 "/[a-zA-Z]"。下面还有几个这样的字符范围：

```
项      匹配                    相当于
\d      数位                    [0-9]
```

\D	非数位	[^0-9]	
\x	十六进制数位	[0-9a-fA-F]	
\X	非十六进制数位	[^0-9a-fA-F]	
\s	空白字符	[ ]	(<Tab> 和 <Space>)
\S	非空白字符	[^]	(非 <Tab> 和 <Space>)
\l	小写字母	[a-z]	
\L	非小写字母	[^a-z]	
\u	大写字母	[A-Z]	
\U	非大写字母	[^A-Z]	

备注：
使用这些预定义的字符范围要比使用它们所表示的那个字符范围快很多。
这些项不能用于 [] 方括号内。因此 "[\d\l]" 不能用来匹配一个数字或一个小写字母。请换用 "\(\d\|\l\)"。

/\s 包括一个这类范围的完整的列表。

27.7 字符类

一个特定的字符范围匹配一组固定的字符。一个字符类与字符范围相似，不过有个本质的区别：一个字符类代表的那组字符可以重新定义而无须改动查找模式。
例如，查找这个模式：

```
/\f\+
```

其中 "\f" 项表示文件名字符。因而这个命令匹配一个由可用作文件名的字符字符组成的序列。
哪些字符可以用来组成文件名取决于你所使用的系统。在微软视窗上，反斜杠可以，而在 Unix 上却不行。文件名字符由 'isfname' 选项来规定。在 Unix 上，该选项的默认值为：

```
:set isfname
isfname=@,48-57,/,.,-,_,+,,,#,$,%,~,=
```

在其它系统上，其默认值各不相同。因此你可以用 "\f" 组成一个查找模式，以匹配一个文件名。该查找模式将自动调整以适应你所使用的系统。

备注：
实际上，Unix 允许在文件名里使用几乎所有的字符，包括空格字符。把这些字符包括在 'isfname' 里，在理论上是没错的。但那样就无法在文本中发现一个文件名在哪儿结束。因此选项 'isfname' 的默认值是个折衷方案。

字符类有如下这些：

项	匹配	选项
\i	标识符字符	'isident'
\I	类似于 \i，但不包括数字字符	
\k	关键词字符	'iskeyword'
\K	类似于 \k，但不包括数字字符	
\p	可显示字符	'isprint'
\P	类似于 \p，但不包括数字字符	
\f	文件名字符	'isfname'
\F	类似于 \f，但不包括数字字符	

27.8 匹配换行符

Vim 能找寻含有换行符的模式。你需要指定换行符在哪儿出现，因为迄今为止所有我们曾经提到过的项，都不匹配换行符。

用 "\n" 项可以在一个特定的位置查验一个换行符：

```
/one\ntwo
```

这个命令将在一行以 "one" 结尾，而下一行以 "two" 开头的地方找到匹配。如果还要匹配 "one two"，那么你需要匹配一个空格或一个换行符。这可以用 "_s" 表示：

```
/one\_stwo
```

若要允许任意数量的空格：

```
/one\_s\+two
```

这个命令也匹配 "one " 在行尾，及 "   two" 在下一行行首的情形。

"\s" 匹配空格字符，"_s" 匹配空格字符或一个换行符。同理，"\a" 匹配一个字母字符，而 "_a" 匹配一个字母字符或一个换行符。其它字符类和字符范围都可以通过插入一个 "_" 来更改其范围。

很多别的项也可以在前面加 "_" 以匹配一个换行符。例如："_." 匹配任意字符或一个换行符。

备注：
"_.*" 匹配任何字符，直至文件结束。要小心，它会使查找命令执行得非常缓慢。

另一个例子是 "_[]"，一个包含了换行符的字符范围：

```
/"\_[^"]*"
```

这个命令找寻位于一对双引号之间，可能分隔成数行的文本。

27.9 举例

这儿有几个查找模式，你也许会觉得有用。本节向你演示如何综合使用前面提及的那些用法。

寻找一个加州驾驶牌照

有一个驾驶牌照号码为 "1MGU103"。它有一个数字，三个大写字母，然后三位数字。直接把这个号码放入查找模式：

```
/\d\u\u\u\d\d\d
```

另一种方法是用一个计数器来指定其中有三个数字和三个字母：

```
/\d\u\{3}\d\{3}
```

换用 [] 字符范围方法：

```
/[0-9][A-Z]\{3}[0-9]\{3}
```

这些方法之中你应该使用哪一种？挑那种你记得住的。你记得住的简单方法要比你记不住的华丽方法快得多。如果你能把它们都记住，那么避免使用最后那种，因为它要打的字多，而且执行起来慢。

寻 找 一 个 标 识 符

在 C 程序里（以及其它很多计算机程序）一个标识符以字母开头，其余部分由字母和数字组成。下划线字符也可以。这样一个标识符可以用下面的命令找到：

```
/\<\h\w*\>
```

"\<" 和 "\>" 用来寻找那些完整的词。"\h" 表示 "[A-Za-z_]"，而 "\w" 则表示 "[0-9A-Za-z_]"。

备注：
"\<" 和 "\>" 取决于 'iskeyword' 选项。如果这个选项包括 "-" 的话，那么 "ident-" 就不匹配了。在这种情况下，请用：

```
/\w\@<!\h\w*\w\@!
```

这个命令查验是否 "\w" 不匹配标识符之前或之后的字符。
参见 /\@<! 和 /\@! 。

下一章： usr_28.txt 折叠

VIM 用户手册 - by Bram Moolenaar
译者：Chimin Yen

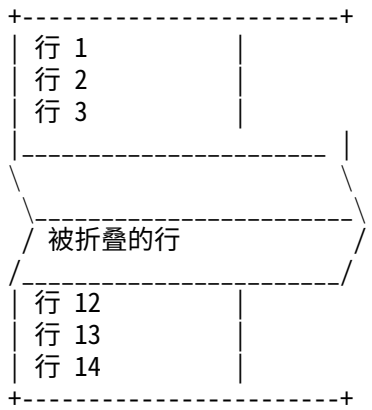
折叠

结构化的文本可以分为许多节。而一节之内可以再分小节。折叠允许你将一节显示为一行，并提供文本的概览。本章将解释各种实现折叠的方法。

下一章：　usr_29.txt　在代码间移动
前一章：　usr_27.txt　查找命令及模式
目录：　usr_toc.txt

28.1　什么是折叠?

折叠用于把缓冲区内某一范围内的文本行显示为屏幕上的一行。就像一张纸，要它缩短些，可以把它折叠起来：



那些文本仍然在缓冲区内而没有改变。受到折叠影响的只是文本行显示的方式。

折叠的好处是，通过把多行的一节折叠成带有折叠提示的一行，会使你更好地了解对文本的宏观结构。

28.2 手动折叠

试一试：把光标置于某一段落内，并键入：

```
zfap
```

你将会看到该段落被一行高亮的文本所代替。你已经创建了一个折叠了。 zf 是个操作符，而 ap 是一个文本对象。你可以将 zf 操作符跟任何一个移动命令联用，为所经之处的文本创建一个折叠。 zf 也能在可视模式下使用。

若要再阅读那些文本，可以键入以下命令以打开该折叠：

```
zo
```

你还可以用以下命令再关闭该折叠：

```
zc
```

所有的折叠命令都以 'z' 开头。展开你的想像力，这个字母看起来就像一张折叠起来的纸的侧面。而 "z" 后面可用的字母，由于采用了帮助记忆方法选择，很容易记得住：

```
zf      F-old creation (创建折叠)
zo      O-pen a fold (打开折叠)
zc      C-lose a fold (关闭折叠)
```

折叠可以嵌套：一个含有折叠的文本区可以被再次折叠。例如，你可以折叠本节内每一段落，然后折叠本章内所有的节。试试看。你将注意到，打开全章的折叠，会将节的折叠还原得跟以前一样，有些打开，而有些关闭。

假定你已经创建了若干折叠，而现在需要阅览全部文本。你可以移到每个折叠处，并键入 "zo"。若要做得更快，可以用这个命令：

```
zr
```

这将减少 (R-educe) 折叠。相反的操作是：

```
zm
```

这将折叠更多 (M-ore)。你可以重复 "zr" 和 "zm" 来打开和关闭若干层嵌套的折叠。

如果你有一个嵌套了好几层深的折叠，你可以用这个命令把它们全部打开：

```
zR
```

这将减少折叠直至一个也不剩。而用下面这个命令你可以关闭所有的折叠：

```
zM
```

这将增加折叠，直至所有的折叠都关闭了。

你可以用 zn 命令快速禁止折叠功能。然后 zN 恢复原来的折叠。 zi 切换于两者之间。以下步骤是一种实用的操作方法：
- 创建折叠，以获取你的文件的概览
- 移动到你要操作的地方

- 执行　zi　以便一边看着文本，一边编辑
- 再执行　zi　　以便移动到另一处

在参考手册中有更多关于手动折叠的内容：　fold-manual

28.3　对折叠的操作

当一些折叠关闭时，移动命令，如 "j" 和 "k" 会轻松的移过折叠，就像它是单个空行一样。这允许你快速移过折叠了的文本。

你可以向对待单行一样复制，删除和粘贴折叠。当你要改动某个程序里的函数的先后次序时，这是很实用的。首先，为 'foldmethod' 选择一个正确的折叠方法，以保证每个折叠包含了整个函数 (或稍缺一点儿)。然后，用 "dd" 命令删除该函数，移动光标，并用 "p" 命令粘贴。如果函数中某些行在折叠之上，或之下，你可以利用可视模式下的选择方法：
- 把光标置于被移文本的首行
- 击 "V" 键开始可视模式
- 把光标置于被移文本的末行
- 击 "d" 键删除被选中的行。
- 把光标移到新位置，并击 "p" 键把文本粘贴在那儿。

有时候，查看或记住一个折叠在哪儿，挺不容易的。更别说用　zo　命令来打开了。要查看那些已定义的折叠：

```
:set foldcolumn=4
```

这个命令将在窗口左边显示一小栏来标识各个折叠。一个 "+" 表示某个关闭的折叠。一个 "-" 表示每个打开的折叠的开头，而 "|" 则表示该折叠内其余的行。

你可以在折叠栏内用鼠标点击 "+"，以打开一个折叠。点击 "-" ，或在它之下的某个 "|" ，将关闭一个打开的折叠。

打开所有光标行上的折叠用　zO 。
关闭所有光标行上的折叠用　zC 。
删除一个光标行上的折叠用　zd 。
删除所有光标行上的折叠用　zD 。

当你进入插入模式后，光标行上的折叠永远不会关闭。那是要让你看见你打的什么字!

当光标前后跳转至折叠，或在折叠上左右移动时，折叠就会自动打开。例如，零命令 "0" 打开光标下的折叠 (假设 'foldopen' 包含 "hor"，即默认设置)。'foldopen' 选项可以修改，为指定的某一类命令打开折叠。如果你要光标遇到折叠，折叠就打开，那么可以这么做：

```
:set foldopen=all
```

警告：你将因此无法移到一个关闭的折叠上。你也许只想临时用一用这个命令，然后把它设回默认值：

```
:set foldopen&
```

你可以在移开折叠时自动关闭折叠：

```
:set foldclose=all
```

这个命令将重新把折叠级别 'foldlevel' 作用到所有的不含光标的折叠。你必须自己试试看你会不会喜欢这个设置。用 zm 增加折叠级别，并用 zr 减少折叠级别（减少折叠的层次）。

折叠是限于本地窗口的。这允许你为同一缓冲区打开两个窗口，一个带折叠，而另一个不带折叠。或者，一个让所有的折叠关闭，而另一个则让所有的折叠打开。

28.4 存储和恢复折叠

当你放弃一个文件时（开始编辑另一个），其折叠状态就丢失了。如果你稍后再回来编辑同一文件，那么，所有手动打开和关闭的折叠，全都恢复到它们的默认状态了。如果折叠是用手动方式创建的，则所有的折叠都消失了！为了保存折叠，可以用 :mkview 命令：

```
:mkview
```

这将储存那些影响文件视图的设定及其它内容。你可以利用 'viewoptions' 选项修改储存的范围。当你稍后回到同一文件时，你可以重新载入这个视图：

```
:loadview
```

你可以为一个文件储存多至十个视图。例如，把当前设置储存为第三个视图，并载入第二个视图：

```
:mkview 3
:loadview 2
```

注意 当你插入或删除一些文本行时，视图可能变得无效。还得检查 'viewdir' 选项，它指定视图文件储存在哪儿。你可能时不时需要删除旧的视图文件。

28.5 依缩进折叠

使用 zf 来定义一个折叠很费事。如果你的文本依循一种结构，以较多的缩进表示较低的层次，那么，你可以采用缩进折叠的方法。这将为每一系列有相同缩进的行创建一个折叠。缩进较多的行将成为嵌套的折叠。缩进折叠适用于许多编程语言。

我们来试试这个方法。先设定 'foldmethod' 选项：

```
:set foldmethod=indent
```

然后你可以用 zm 和 zr 命令增加和减少折叠。在下面这个例文上很容易看明白：

```
本行没有缩进
        本行被缩进一次
                本行被缩进两次
                本行被缩进两次
        本行被缩进一次
本行没有缩进
        本行被缩进一次
```

本行被缩进一次

注意 缩进多少和折叠深度之间的关系倚赖于 'shiftwidth' 选项。每个 'shiftwidth' 选项规定的缩进宽度，在折叠深度上加一。这被称为一个折叠级别。

当你使用 zr 和 zm 命令时，你实际上是在增加或减少 'foldlevel' 选项。你也可以直接设置它：

```
:set foldlevel=3
```

这意味着，所有缩进等于或大于 'shiftwidth' 三倍的折叠将被关闭。折叠级别设定得越低，越多的折叠将被关闭。当 'foldlevel' 为零时，所有的折叠都将被关闭。 zM 把 'foldlevel' 设为零。相反的命令 zR 把 'foldlevel' 设为文件中最深的折叠级别。

因此，有两种方法开启和关闭折叠：
(A) 设定折叠级别。
    这提供了一种极快的 "缩小" 方法来查看文本结构，移动光标，以及重新 "放大" 到具体的文本。

(B) 利用 zo 和 zc 命令打开和关闭指定的折叠。
    这个方法允许你仅仅打开那些你要打开的折叠，而不影响其它的折叠。

这两种方法可以结合起来用：你可以先用几次 zm 以关闭大多数折叠，然后用 zo 打开一个指定的折叠。或者，用 zR 打开所有的折叠，然后用 zc 关闭指定的折叠。

但是，当折叠方法 'foldmethod' 的值为 "indent" 时，你不能手动定义折叠。因为那样会引起缩进宽度和折叠级别之间的冲突。

在参考手册中有更多关于缩进折叠的内容： fold-indent

28.6 依标志折叠

文本中的标志用于指定一个折叠区的起点和终点。标志折叠可以精确地控制一个折叠究竟包含哪些行文本。缺点是文本需要改动。

试试这个：

```
:set foldmethod=marker
```

以下列 C 程序片段为例：

```
/* foobar () {{{ */
int foobar()
{
        /* return a value {{{ */
        return 42;
        /* }}} */
}
/* }}} */
```

请 注意 ，折叠行将显示位于标志之前的文字。这正好用来说明该折叠包含了什么。

令人十分困扰的是，当某些文本行移动后，标志不再正确地配对。这种局面可以利用编号

标志来避免。例如：

```
/* global variables {{{1 */
int varA, varB;

/* functions {{{1 */
/* funcA() {{{2 */
void funcA() {}

/* funcB() {{{2 */
void funcB() {}
/* }}}1 */
```

每一个编号标志表示一个编号指定级别的折叠的开始。这将使任何较高层次的折叠在此结束。你可以只用编号标志的开始符定义所有的折叠。只有当你要明确地在另一个开始前结束一个折叠时，你才需要加一个标志停止符。

在参考手册中有更多关于标志折叠的内容： fold-marker

28.7 依语法折叠

Vim 为每一种不同的语言使用一个不同的语法文件。语法文件为文件中各种不同语法项定义颜色。如果你正用 Vim 在一个支持色彩的终端上阅读本文，你所看见的色彩就是由语法文件 "help" 定制的。

在语法文件中，你可以加入一些带有 "fold" 参数的语法项。这些语法项将定义折叠区。这要求写一个语法文件，把这些项目加入其中。编写这样一个文件是不容易的。但是一旦写成，所有折叠的创建就变成自动的了。

在此，我们将假定你正使用一个已经写好的语法文件。这样的话，就没更多解释的必要了。你可以像以上解释过的那样打开和关闭折叠。编辑文件时折叠会自动创建和删除。

在参考手册中有更多关于语法折叠的内容： fold-syntax

28.8 依表达式折叠

表达式折叠类似于缩进折叠，但并非利用文本行的缩进，而是调用一个函数来计算一行的折叠级别。当文本的一部分表明那些行属于同一组时，你可以使用这个方法。一个例子是电子邮件，其中引述的文本由行首的 ">" 来表示。要折叠这些引文，可以用以下命令：

```
:set foldmethod=expr
:set foldexpr=strlen(substitute(substitute(getline(v:lnum),'\\s','',\"g\"),'[^>].*','',''))
```

你可以在这段文本上试式看：

```
> quoted text he wrote
> quoted text he wrote
> > double quoted text I wrote
> > double quoted text I wrote
```

以下是上例中 'foldexpr' 的解释 (自里至外)：

```
getline(v:lnum)                    读取当前行
substitute(...,'\\s','','g')       从当前行删除所有空白字符
```

```
substitute(...,'[^>].*','','')      删除行首那些 '>' 之后的任何字符
strlen(...)                         计算字符串的长度，即 '>' 的个数
```

注意 在 ":set" 命令中，每一个空格，双引号和反斜线符之前，必须插入一个反斜杠。如果这会把你搞糊涂，那么就执行

```
:set foldexpr
```

来检查所产生的实际值。为了修正一个复杂的表达式，请使用命令行补全:

```
:set foldexpr=<Tab>
```

其中 <Tab> 是一个真实的 Tab 键。Vim 将填入以前的值，然后你可以编辑它。

当该表达式变得相对复杂时，你应当将其放入一个函数。然后设定 'foldexpr' 来调用该函数。

在参考手册中有更多关于表达式折叠的内容： fold-expr

28.9 折叠未被改动的行

当你在同一窗口也设定 'diff' 选项时，这种折叠方法就很有用。 vimdiff 命令为你设定好了使用未改行折叠。例如:

```
:setlocal diff foldmethod=diff scrollbind nowrap foldlevel=1
```

在显示同一文件的不同版本的每个窗口内执行这个命令。你将清楚地看到不同文件间的区别，因为那些没改动的文本都被折叠起来了。

更多细节参见 fold-diff 。

28.10 使用哪种折叠办法呢?

所有这些可能性让你感到纳闷，你究竟应该选择哪种方法。不幸的是，没有放之四海皆准的法则。这里只给出一些提示。

如果存在一个语法文件，其中定义的折叠符合你正在使用的程序语言，那么，语法折叠应该是最好的选择。否则，你也许得试着写一个。这要求你相当的了解关于查找模式知识。这并非易事。但一旦写成，你将不必以手动的方式定义折叠了。

键入命令，手动折叠一个个文本区的方法可用于无结构特点的文本。然后用 :mkview 命令来储存和还原折叠状态。

标志折叠法要求你修改文件。如果你与它人共享这个文件，或不得不遵守公司规定的标准，那么你也许得不到许可给文件加标志。

标志折叠的主要优点是，你可以精确的把标志放在你要的位置。那样就避免了那种在你剪切和粘贴折叠时漏了几行文本的情况。并且，你还可以加个注释，说明该折叠包含些什么。

缩进折叠法是那种在许多文件里都用的着。但并不是每次都能成功的方法。当你无法采用其它方法时，就用这种。然而，缩进折叠在做提纲时特别有用。你必须为每一层嵌套折叠

特意的使用同一缩进宽度 'shiftwidth'。

表达式折叠法能够在几乎任何有结构特定的文本中创建折叠。这种方法相当简单，尤其当折叠的开始和结束处能容易地被识别的时候。
    如果你用 "expr" 方法来定义折叠而无法得到完全满意的结果。那么你可以切换到手动方法 "manual"。这么做不会删除那些已经定义好了的折叠。之后你便可以手动删除或增加折叠了。

下一章: usr_29.txt   在代码间移动

usr_29.txt　　适用于 Vim 9.1 版本。　　最近更新：2025年3月

VIM 用户手册 - by Bram Moolenaar
译者：wandys

在代码间移动

Vim 的创造者是一名计算机程序员，因此这就不奇怪 Vim 中有许多帮助编写程序的功能：跳转到标识符被定义和使用的地方；在另一个窗口中预览有关的声明等等。在下一章中还会介绍更多的功能。

下一章：　usr_30.txt　　编辑程序
前一章：　usr_28.txt　　折叠
　目录：　usr_toc.txt

29.1　使用标签

什么是标签？标签就是一个标识符被定义的地方。一个例子就是 C 或者 C++ 程序中的函数定义。标签列表可以保存在一个标签文件中。Vim 可以通过它来从任何地方跳转到该标签，也就是一个标识符被定义的地方。
　在当前目录下为所有的 C 文件生成标签文件，使用下面的这个命令：

```
ctags *.c
```

"ctags" 是一个独立的程序。大多数 Unix 系统上都已经安装了它。如果你还没有安装，可以在这里找到 Universal/Exuberant ctags:

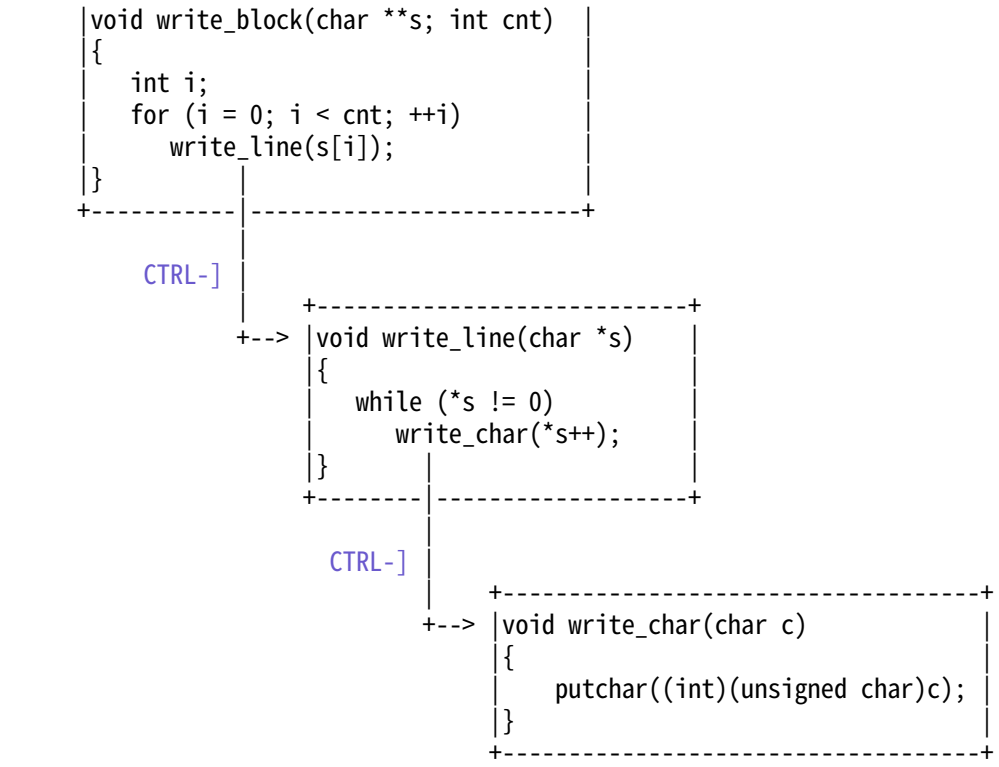
推荐使用 Universal ctags，Exuberrant ctags 不再开发了。

现在你可以使用下面的命令跳转到一个函数定义的地方：

```
:tag startlist
```

这个命令会找到函数 "startlist"，即使该函数是在另一个文件中。
　CTRL-] 命令会跳转到当前光标下单词的标签。这样浏览毫无头绪的 C 代码会变得更容易些。举个例子，假设你在函数 "write_block" 中。你可以看到它调用了函数 "write_line"。但 "write_line" 做了什么呢？将光标置于调用 "write_line" 的地方然后按 CTRL-]，你就跳转到了这个函数的定义的地方了。
　"write_line" 函数调用了 "write_char"。你需要知道它做了什么。将光标定位到调用 "write_char" 的地方然后按 CTRL-]，你就到了定义 "write_char" 的地方。

```
+-----------------------------------+
```



":tags" 命令显示你经过的标签列表:

```
:tags
  # TO tag         FROM line  in file/text
  1  1 write_line          8  write_block.c
  2  1 write_char          7  write_line.c
>
```

现在介绍向回跳转。CTRL-T 命令跳转到上一个标签。在上例中，你会回到 "write_line" 函数调用 "write_char" 的地方。

这个命令接受一个计数参数，用来表示跳转回去的标签个数。你已经向前跳转，现在又跳转了回去。现在我们再一次向前跳转。下面的命令跳转到标签列表中最上面的标签:

```
:tag
```

你可以在前面加上要向前跳转的标签个数。比如: ":3tag"。CTRL-T 同样可以加上一个计数参数。

通过这些命令，你可以用 CTRL-] 沿着调用树向前跳转，用 CTRL-T 向回跳转，用 ":tags" 命令显示当前位置。

分 割 窗 口

":tag" 命令会将当前窗口的文件替换为包含新函数的文件。怎样才能同时查看两个文件呢? 你可以使用 ":split" 命令将窗口分开然后再用 ":tag" 命令。Vim 有个缩写命令可以做到这些:

```
:stag tagname
```

使用下面的命令可以分割当前窗口并跳转到光标下的标签：

```
CTRL-W ]
```

如果指定了计数参数，新窗口将包含指定的那么多行。

多 个 标 签 文 件

如果在多个目录中都有文件，你可以在每一个目录下创建一个标签文件。Vim 只能跳转到那个目录下的标签。

通过设定 'tags' 选项，你可以使用多个相关的标签文件。比如：

```
:set tags=./tags,./../tags,./*/tags
```

这会使 Vim 找到当前文件所在目录及其父目录和所有子目录下的标签文件。

这已经是不少的标签文件了，但也许仍不够。比如，当编辑 "~/proj/src" 目录下的一个文件时，你无法找到 "~/proj/sub/tags" 目录下的标签文件。对这种情况，Vim 提供了一个查找整个目录树下标签文件的方法，比如：

```
:set tags=~/proj/**/tags
```

单 个 标 签 文 件

当 Vim 在多个地方查找标签文件时，你会听到硬盘在格格作响。这样会有点慢。在这种情况下，你最好将这些时间花在生成一个大的标签文件上。你可以要等一会儿。

这得借助上面提到的 Universal 或 Exuberant ctags 程序。它有一个选项可以搜索整个目录树：

```
cd ~/proj
ctags -R .
```

这样做的好处是 Universal/Exuberant ctags 可以识别多种文件类型，它不仅适用于 C 和 C++ 程序，还适用于 Eiffel 甚至 Vim 脚本。请参考 ctags 文档进行调整所用参数。

现在你只需要告诉 Vim 你的标签文件在何处：

```
:set tags=~/proj/tags
```

多 个 匹 配

当一个函数（或类中的方法）被定义多次，":tags" 命令会跳转到第一处。如果在当前文件中存在匹配，那它将会被首先使用。

你现在可以跳转到同一个标签的其它匹配处：

```
:tnext
```

重复执行这个命令可以找到更多的匹配。如果存在很多匹配，你可以选择要跳转到哪一个：

```
:tselect tagname
```

Vim 会为你展示一个选择列表：

```
  # pri kind tag               file
  1 F   f    mch_init          os_amiga.c
               mch_init()
  2 F   f    mch_init          os_mac.c
               mch_init()
  3 F   f    mch_init          os_msdos.c
               mch_init(void)
  4 F   f    mch_init          os_riscos.c
               mch_init()
Enter nr of choice (<CR> to abort):
```

你现在可以输入要跳转到的匹配代号（在第一列）。其它列的信息可以让你知道匹配在何处被定义。

可以用这些命令在各匹配的标签间移动：

```
:tfirst                 到第一个匹配
:[count]tprevious       向前 [count] 个匹配
:[count]tnext           向后 [count] 个匹配
:tlast                  到最后一个匹配
```

如果没有指定，[count] 缺省为一。

猜测标签名

命令行补全是避免输入长标签名的好办法。只需输入开始的一部分然后按 <Tab>：

```
:tag write_<Tab>
```

你会得到第一个匹配。如果这不是你想要的，重复输入 <Tab> 直到你找到正确的匹配。

有时你只知道一个函数名的一部分，或是你有很多以相同字符串开头而结尾不同的标记。这时你可以告诉 Vim 使用一个模式来查找标签。

假设你要跳转到一个包含 "block" 的标签。首先输入：

```
:tag /block
```

现在再利用命令行补全功能：输入 <Tab>。Vim 会找到所有包含 "block" 的标签并使用第一个匹配。

标签名前面的 "/" 告诉 Vim 这不是一个确定的标签名而是一个模式。你可以利用有关查找模式的所有特性。举个列子，假设你要选择所有以 "write_" 开头的标签：

```
:tselect /^write_
```

"^" 指定标签以 "write_" 开头，否则在中间含有 "write_" 的标签名也会被找到。类似地，"$" 指定标签名结尾处的匹配。

标签浏览器

CTRL-] 可以让你跳转到光标所在标识符的定义处，因此你可以利用标识符的列表来形成

一个目录。这里给出一个例子。
首先生成一个标识符列表 (需要 Universal 或 Exuberant ctags):

```
ctags --c-types=f -f functions *.c
```

现在打开 Vim 并在一个垂直分割窗口中编辑这个文件:

```
vim
:vsplit functions
```

窗口中包含一个所有函数的列表。其它的东西可以被忽略。用 ":setlocal ts=99" 命令使其显示得更清晰些。
在这个窗口中，定义一个映射:

```
:nnoremap <buffer> <CR> 0ye<C-W>w:tag <C-R>"<CR>
```

移动光标至要跳转到函数的所在行，输入 <Enter>。Vim 会在另一个窗口中跳转到所选择的函数定义处。

相 关 杂 项

你可以在不改变 'tagcase' 为 "followic" 的情况下置位 'ignorecase' 选项，或者设置 'tagcase' 为 "ignore" 来忽略标签名里的大小写。

'tagbsearch' 选项标明标签文件是否经过排序。缺省是假定为标签文件已排序，这样会使查找更快，但如果文件没有被排序是无法工作的。

'taglength' 选项可用来告诉 Vim 标签的有效字符个数。

Cscope 是一个自由软件。它不仅可以找到一个标识符被声明的地方，还可以找到标识符被使用的地方。请参考 cscope 。

29.2 预览窗口

当编辑含有函数调用的代码时，你需要使用正确的调用参数。要获知所要传递的值，你可以查看这个函数是如何定义的。标签机制对此十分适用。如果定义可在另一个窗口内显示那就更好了。对此我们可以利用预览窗口。
打开一个预览窗口来显示函数 "write_char":

```
:ptag write_char
```

Vim 会打开一个窗口，跳转到 "write_char" 标签。然后它会回到原来的位置。这样你可以继续输入而不必使用 CTRL-W 命令。
如果函数名出现在文本中，你可以用下面的命令在预览窗口中得到其定义:

```
CTRL-W }
```

有一个脚本可以自动显示光标处的标签定义。请参考 CursorHold-example 。

用下面的命令关闭预览窗口:

```
:pclose
```

要在预览窗口中编辑一个指定的文件，用 ":pedit"。这在编辑头文件时很有用，比如：

```
:pedit defs.h
```

最后，"psearch" 可用来查找当前文件和任何包含文件中的单词并在预览窗口中显示匹配。这在使用没有标签文件的库函数时十分有用。例如：

```
:psearch popen
```

这会在预览窗口中显示含有 popen() 原型的 "stdio.h" 文件：

```
FILE    *popen __P((const char *, const char *));
```

你可以用 'previewheight' 选项指定预览窗口打开时的高度。

29.3 在代码间移动

因为程序代码是结构化的，Vim 可以识别其中的有关项目。一些特定的命令可用来完成相关的移动。

C 程序中经常包含类似下面的代码：

```
#ifdef USE_POPEN
    fd = popen("ls", "r")
#else
    fd = fopen("tmp", "w")
#endif
```

有时会更长，也许还有嵌套。将光标置于 "#ifdef" 处按 %。Vim 会跳转到 "#else"。继续按 % 会跳转到 "#endif"。再次按下 % 又回到原来的 "#ifdef"。

当代码嵌套时，Vim 会找到相匹配的项目。这是检查你是否忘记了一个 "#endif" 的好办法。

当你在一个 "#ifdef" - "#endif" 块内的某个位置，你可以用下面的命令回到开始处：

```
[#
```

如果你的位置不是在 "#if" 或 "#ifdef" 之后，Vim 会鸣音。用下面命令可以跳转到下一个 "#else" 或 "#endif"：

```
]#
```

这两个命令会跳过它所经过的 "#if" - "#endif" 块。
例如：

```
#if defined(HAS_INC_H)
    a = a + inc();
# ifdef USE_THEME
    a += 3;
# endif
    set_width(a);
```

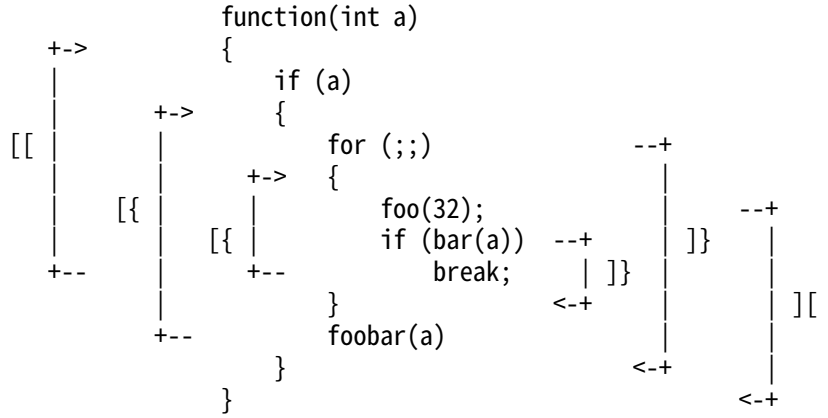
如果光标在最后一行，"[#" 会移动到第一行。中间的 "#ifdef" - "#endif" 块被跳过。

在代码块内移动

C 代码块包含在 {} 中，有时一个代码会很长。要跳转到外部代码块的开始处，用 "[[" 命令。用 "][" 找到结尾处。(前提是 "{" 和 "}" 都在第一列。)

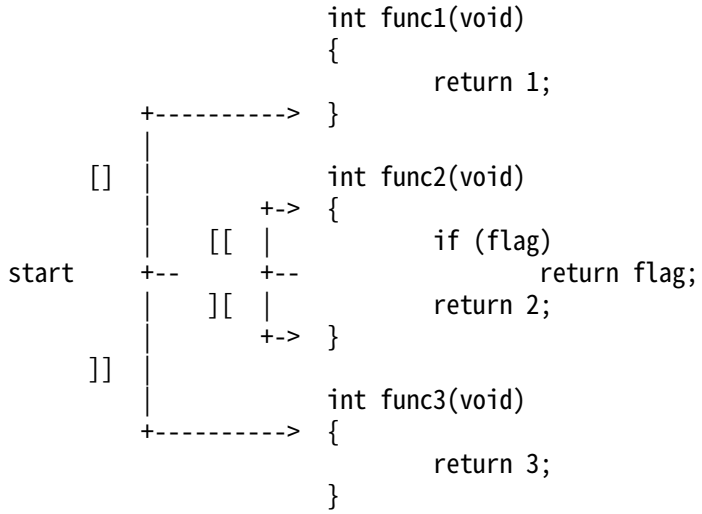
"[{" 命令跳转到当前代码块的开始处。它会跳过同一级别的 {} 对。"]}" 跳转到结尾。

一点概述：



当编写 C++ 或 Java 代码时，外部代码块是类，而下一级的 {} 是方法。在类内部用 "[m" 可以找到前一个方法的开始。"]m" 会找到下一个方法的开始。

另外，"[]" 反向移动到前一个函数的结尾，"]]" 正向移动到下一个函数的开始。函数的结尾指的是处在第一列的 "}"。



不要忘了你还可以用 "%" 在匹配的 ()、{} 和 [] 间移动。这在它们相距很多行时仍然适用。

在括号内移动

"[(" 和 "])" 命令与 "[{" 和 "]}" 类似，只不过它们适用于 () 对而不是 {} 对。

```
                        [(
                <--------------------------------
                          <-------
            if (a == b && (c == d || (e > f)) && x > y)
                          ------------->
                  ------------------------------->
                                ])
```

在 注 释 间 移 动

移动到一个注释的开始用 "[/"；向前移动到注释的结尾用 "]/"。这只对 /* - */ 注释有效。

```
   +->     +-> /*
   |    [/ |    * A comment about      --+
[/ |       +--  * wonderful life.        | ]/
   |            */                     <-+
   |
   +--          foo = bar * 3;          --+
                                          | ]/
                /* a short comment */   <-+
```

29.4 查找全局标识符

你在编辑一个 C 程序，想要知道一个变量是被声明为 "int" 还是 "unsigned"。一个快速的方法是使用 "[I" 命令来查找。

假设光标在单词 "column" 处。输入：

```
[I
```

Vim 会列出它所找出的匹配行，不仅在当前文件内查找，还会在所有的包含文件中查找。结果如下所示：

```
structs.h
 1:   29     unsigned     column;    /* column number */
```

相对使用标签文件或预览窗口的好处是包含文件也被搜索。大多数情况下都能找到正确的声明。即使标签文件已经过期或者你没有为包含文件建立标签也不会影响结果。

但是一些准备工作是必要的，否则 "[I" 就没法工作。首先，'include' 选项必须指定文件是如何被包含的。缺省值适用于 C 和 C++。对其它的语言，你需要自己设定。

定 位 包 含 文 件

Vim 会找到 'path' 选项指定路径中的包含文件。如果缺少某个目录，一些包含文件将不会被找到。你可以用这个命令来查看：

```
:checkpath
```

它会列出不能找到的包含文件，以及被找到的包含文件。一个输出样例：

```
--- Included files not found in path ---
<io.h>
vim.h -->
  <functions.h>
  <clib/exec_protos.h>
```

文件 "io.h" 被当前文件包含但无法找到。"vim.h" 可以找到，这样 ":checkpath" 跟进这个文件并检查其中的包含文件。结果显示无法找到 "vim.h" 包含的 "functions.h" 和 "clib/exec_protos.h" 文件。

备注:
Vim 不是一个编译器。它无法识别 "#ifdef" 语句。这就是说所有的 "#include" 语句都会被使用，即使它在 "#if NEVER" 之后。

给 'path' 选项增加一个目录可以修正无法找到文件的错误。一个好的参考是 Makefile。注意那些包括 "-I" 的条目，比如 "-I/usr/local/X11"。要增加这个目录，用:

```
:set path+=/usr/local/X11
```

如果有很多的子目录，你可以用 "*" 通配符。例如:

```
:set path+=/usr/*/include
```

这会找到 "/usr/local/include" 以及 "/usr/X11/include" 目录下的文件。

如果你的工程项目的包含文件都在一个嵌套的目录树下，"**" 就非常有用。它会搜索所有的子目录。例如:

```
:set path+=/projects/invent/**/include
```

这会找到这些目录下的文件:

```
/projects/invent/include
/projects/invent/main/include
/projects/invent/main/os/include
等等
```

还有其它的可能性。更多信息，请查看 'path' 选项。
如果你想查看找到的包含文件，用这个命令:

```
:checkpath!
```

你会得到一个 (很长) 的包含文件列表。为使它更短些，Vim 会对已经找到的文件显示 "(Already listed)" 而不再重新显示一遍。

跳 转 到 匹 配

"[I" 产生一个每项只有一行文本的列表。如果你想要进一步地查看第一项，你可以用这个命令来跳转:

```
[<Tab>
```

你也可以使用 "[CTRL-I"，因为 CTRL-I 和按 <Tab> 效果一样。

"[I" 产生的列表在每行的开头都有一个序号。如果你要跳转到第一项外的其它项，首先输入序号：

```
3[<Tab>
```

会跳转到列表中的第三项。记住你可以用 CTRL-O 跳回到原来的地方。

相 关 命 令

```
[i              只列出第一项匹配
]I              只列出光标下面的项目
]i              只列出光标下面的第一项匹配
```

查 找 宏 定 义 标 识 符

"[I" 命令查找任何标识符。只查找 "#define" 定义的宏，用：

```
[D
```

同样，这会在所有的包含文件中查找。'define' 选项指定 "[D" 所查找的项目定义行的模式。你需要改变它的值来适用于 C 或 C++ 以外的语言。
"[D" 相关命令：

```
[d              只列出第一项匹配
]D              只列出光标下面的项目
]d              只列出光标下面的第一项匹配
```

29.5 查找局部标识符

"[I" 命令查找所有的包含文件。要在当前文件中查找并跳转到光标处单词被首次使用的地方，用：

```
gD
```

提示：Goto Definition。这个命令对查找局部 (C 语言中的 "static") 声明的变量或函数很有用。例如 (光标在 "counter" 处)：

```
   +->   static int counter = 0;
   |
   |     int get_counter(void)
gD |     {
   |         ++counter;
   +--       return counter;
             }
```

要进一步的缩小查找范围，只在当前函数内查找，用这个命令：

```
gd
```

这会回到当前函数的开始处寻找光标处单词首次出现的地方。实际上，它是向后找到一个在第一列为 '{' 的上方的空行，然后再从那里开始正向查找标识符。例如（光标位于 idx 上）：

```
                int find_entry(char *name)
                {
           +->      int idx;
           |
        gd |        for (idx = 0; idx < table_len; ++idx)
           |            if (strcmp(table[idx].name, name) == 0)
           +--              return idx;
                }
```

下一章： usr_30.txt 编辑程序

usr_30.txt　　适用于 Vim 9.1 版本。　　最近更新：2025年3月

VIM 用户手册 - by Bram Moolenaar
译者：wandys

编辑程序

Vim 有很多帮助编写程序代码的命令。例如：直接在 Vim 内编译程序并跳转到出错位置；根据语言种类自动设定缩进，还有对注释进行排版。

下一章：　usr_31.txt　使用 GUI
前一章：　usr_29.txt　在代码间移动
目录：　usr_toc.txt

30.1　编译

Vim 有个 "快速修复" 命令集。通过这些命令，你可在 Vim 内编译程序并能直接跳转到出错位置进行修正。你可以接着重新编译并做修正，直到不再出错为止。

下面的命令运行 "make"（包括你所给出的参数）程序并捕捉其运行结果：

```
:make {arguments}
```

如果有错误产生那么它们将被捕获到。而且光标会停留在出现第一个错误的地方。
　让我们看一个 "make" 的例子。当输入 ":make" 后，屏幕出现如下的结果：

```
:!make | &tee /tmp/vim215953.err
gcc -g -Wall -o prog main.c sub.c
main.c: In function 'main':
main.c:6: too many arguments to function 'do_sub'
main.c: At top level:
main.c:10: parse error before '}'
make: *** [prog] Error 1

2 returned
"main.c" 11L, 111C
(3 of 6): too many arguments to function 'do_sub'
Press ENTER or type command to continue
```

你会看到编译 "main.c" 文件（见下）出现的错误。按一下 <Enter>，Vim 会显示文件 "main.c" 并将光标移到第一个出现错误行。你无需指定文件名或具体的行号，Vim 会根据出错信息定位相关位置。

```
                +-------------------------------------------------+
                |int main()                                       |
                |{                                                |
                |       int i=3;                                  |
        光标 -> |       do_sub("foo");                            |
                |       ++i;                                      |
                |       return (0);                               |
                |}                                                |
                |}                                                |
                | ~                                               |
                |(3 of 12): too many arguments to function 'do_sub' |
                +-------------------------------------------------+
```

这个命令将光标移到下一个错误所在的行:

```
:cnext
```

光标跳转到文件的最后一行（第十行），那里有一个多余的 '}'。
  有时由于空间不够，Vim 会缩短出错信息。如果你想查看详细信息，可执行:

```
:cc
```

你可以通过 ":clist" 命令得到一个有关所有出错信息的概览。输出的内容如下所示:

```
:clist
3 main.c: 6:too many arguments to function 'do_sub'
5 main.c: 10:parse error before '}'
```

只有那些含有文件名或行数的错误信息会被显示。Vim 假定你对其它的信息不感兴趣。有时你也许需要查看那些并不含文件名或行数的信息（比如连接器输出的有关未定义函数的错误信息)。要查看所有信息，在命令后加上 "!" 就可以:

```
:clist!
1 gcc -g -Wall -o prog main.c sub.c
2 main.c: In function 'main':
3 main.c:6: too many arguments to function 'do_sub'
4 main.c: At top level:
5 main.c:10: parse error before '}'
6 make: *** [prog] Error 1
```

Vim 会高亮显示当前错误。如果你想回到上一个错误，执行:

```
:cprevious
```

以下是一些其它的跳转命令:

```
:cfirst         到第一处错误
:clast          到最后一处错误
:cc 3           到第三处错误
```

指 定 编 译 器

通过 'makeprg' 选项可以指定 ":make" 命令启动的程序名。通常它被设定为 "make"。但 Visual C++ 的用户需要将它设定为 "nmake":

```
:set makeprg=nmake
```

你可以对这个选项指定参数。如有特殊字符，请用 '\' 来转义。比如:

```
:set makeprg=nmake\ -f\ project.mak
```

你还可以使用特殊的 Vim 内部关键字。比如用 '%' 来代替当前文件名:

```
:set makeprg=make\ %:S
```

这样，当你编辑文件 "main.c" 时，运行 ":make" 命令会运行下面的程序:

```
make main.c
```

这不太有用，所以你需要用 :r (root) 修饰符来稍稍修改一下:

```
:set makeprg=make\ %:r:S.o
```

现在运行的程序就成了:

```
make main.o
```

有关修饰符的更多信息，请查看这里: filename-modifiers 。

旧 的 出 错 信 息 列 表

假设你用 ":make" 编译了一个程序。其中的一个文件里有个 警告 (warning) 信息而另一个文件中有一个出错 (error) 信息。你修改了这个错误，并再次运行 ":make" 以参看它是否已被纠正。现在你想查看刚才的那个 警告 信息。但由于含有 警告 的那个文件并没有被重新编译，你无法在当前出错信息列表中看到原来的 警告 信息。在这种情况下，你可以用下面的命令来查看上一个出错信息列表:

```
:colder
```

然后你可以通过 ":clist" 和 ":cc {nr}" 命令来跳转到出现 警告 的地方。
要查看下一个出错列表:

```
:cnewer
```

Vim 可以记住十个出错信息列表。

更 换 编 译 器

要做到这一点，你需要告诉 Vim 所使用编译器产生的错误信息格式。这可以通过 'errorformat' 来设定。这个选项几乎可以配合任意一个编译器的使用，但它的具体配置却很复杂。请在这里查看详细解释: errorformat 。

你可能会用到多种不同的编译器。每次都设定 'makeprg' 选项，尤其是 'errorformat'

选项是很繁杂的。为此，Vim 提供一个简便的方法。比如：要使用微软的 Visual C++ 编译器：

```
:compiler msvc
```

这个命令会找到适合 "msvc" 编译器的 Vim 脚本文件并设定相关选项。
你可以为编译器编写自己的脚本文件。请参考 write-compiler-plugin 。

输 出 重 定 向

"make" 命令会将运行结果重定向到一个错误文件中。具体的工作方式取决于很多方面，比如 'shell' 选项。如果你的 ":make" 命令不能捕获输出，请检查 'makeef' 和 'shellpipe' 选项。选项 'shellquote' 和 'shellxquote' 可能也会起作用。

如果你无法利用 ":make" 命令重定向输出，一种变通的方法是在另一个窗口编译程序并将输出重定向到一个文件中。然后你可在 Vim 中查看此文件：

```
:cfile {filename}
```

这样，你就可以像运行 ":make" 命令那样跳转到出错的地方。

30.2 C 风格文件缩进

合理的缩进会使程序更容易理解。Vim 提供了多种方法来简化这项工作。要对 C 或 C 风格如 Java 或 C++ 的程序缩进，请设定 'cindent' 选项。Vim 相当地了解 C 程序，它会尽可能地为你自动缩进。通过 'shiftwidth' 选项，你可以指定下一级的缩进空格数。4 个空格的效果很好。用一个 ":set" 命令就可做到：

```
:set cindent shiftwidth=4
```

设定了这一选项之后，当你输入了一个语句，比如 "if (x)" 后，下一行会自动向下一级缩进。

```
                              if (flag)
自动缩进            --->          do_the_work();
自动取消缩进        <--       if (other_flag) {
自动缩进            --->          do_file();
保持缩进                          do_some_more();
自动取消缩进        <--       }
```

当你在大括号（{}）内输入时，语句会在开始处缩进，而在结束处取消缩进。每次输入 '}' 后都会取消缩进，因为 Vim 不知道你下一步将要输入什么。

自动缩进还能帮助你提前发现代码中的错误。比如当你输入了一个 '}' 后，如果发现比预想中的缩进多，那可能缺少了一个 '}'。请用 "%" 命令查找与你输入的 '}' 相匹配的 '{'。
缺少 ')' 和 ';' 同样会引起额外的缩进。当你发现比预料中多空白时，最好检查一下之前的代码。

当你的代码没有被正确地排版，或者你插入或删除了某些行时，你需要重新进行代码缩进。操作符 "=" 能完成这项功能，最简单的格式是：

```
==
```

这会缩进当前行。像其它的操作符一样，有三种方式使用它。可视模式下，"=" 缩进选中的行。一个有用的文本对象是 "a{"。它会选中当前 {} 区。这样，要重新缩进光标所在代码块：

```
=a{
```

如果原来代码的缩进很糟糕，你还可以重新缩进整个文件：

```
gg=G
```

但是，不要对已经手工缩进的文件使用此命令。自动缩进的确做得很好，但在某些情况下你也许确实需要手工缩进。

设 定 缩 进 风 格

不同的人有不同的缩进风格。在默认情况下，Vim 采用了 90% 的程序员都使用的一种方式并能很好地完成工作。但是，如果你想使用其它的风格，你可以通过 'cinoptions' 选项来设定。

'cinoptions' 默认为空，Vim 会使用默认风格。你可以在你希望改变的地方添加相应的项目。例如，要使大括号的缩进如下所示：

```
if (flag)
  {
    i = 8;
    j = 0;
  }
```

请使用这个命令：

```
:set cinoptions+={2
```

还有很多其它的项目可供使用。请参考 cinoptions-values 。

30.3 自动缩进

你无需每次编辑 C 文件时都手工设定 'cindent' 选项。通过下面的命令你可以使它自动化：

```
:filetype indent on
```

实际上，它不仅为 C 文件打开了 'cindent' 选项。首先，它会使 Vim 自动检查文件类型。语法高亮同样需要此功能。

一旦文件类型被识别，Vim 会为此类型的文件查找相关的缩进文件。(Vim 的发行中包含了适合多种不同编程语言的缩进文件。) 该缩进文件将会被用来缩进当前文件。

如果你不喜欢这项功能，可以将它关闭：

```
:filetype indent off
```

如果你不想为某种特定类型的文件进行缩进，你可以这样做：

首先建一个只包括下行的文件：

```
:let b:did_indent = 1
```

然后将其重命名为：

```
{directory}/indent/{filetype}.vim
```

{filetype} 是文件类型的名字，比如 "cpp" 或 "java"。你可以用下面的命令来得到 Vim 识别到的文件类型名：

```
:set filetype
```

对本文件，输出会是：

```
filetype=help
```

这样你就可以用 "help" 来表示 {filetype}。
对 {directory} 部分，你需要根据你的运行时目录来设定。请查看下面命令的输出：

```
set runtimepath
```

请使用第一项（也就是第一个逗号前的名字）。如果上面命令的输出是：

```
runtimepath=~/.vim,/usr/local/share/vim/vim60/runtime,~/.vim/after
```

你需要使用 "~/.vim" 来表示 {directory}。这样最后的文件名就是：

```
~/.vim/indent/help.vim
```

除了关闭缩进选项，你还可以编写自己的缩进文件。请参考 indent-expression 。

30.4 其它缩进

最简单的自动缩进通过 'autoindent' 选项来完成，它会延续上一行的缩进。稍微聪明点的是 'smartindent'，这个选项对那些没有缩进文件可用的编程语言很有用。'smartindent' 选项没有 'cindent' 选项聪明，但要比 'autoindent' 聪明些。
如果 'smartindent' 被设定，会在每个 '{' 处新增一级缩进，并在每个 '}' 处消减。另外，对于 'cinwords' 选项所设定的所有单词也会添加新一级的缩进。所有以 '#' 开始的行都会被特殊处理：所有缩进都被清除。这样做是为了保持所有的预处理命令都在第一列开始。缩进会在下一行中恢复。

修 正 缩 进

当你利用 'autoindent' 和 'smartindent' 选项延续上一行的缩进时，有很多时候你都需要添加或删除一个 'shiftwidth' 宽度的缩进。一个快速的方法是在插入模式下利用 CTRL-D 和 CTRL-T 命令。
比如，当你需要输入以下的外壳脚本时：

```
if test -n a; then
   echo a
```

```
   echo "-------"
fi
```

设定了这样的选项：

```
:set autoindent shiftwidth=3
```

你先输入了第一行，按下回车后又输入了第二行的开头：

```
if test -n a; then
echo
```

这时你会发现你需要一个额外的缩进。输入 CTRL-T，结果变为：

```
if test -n a; then
   echo
```

在插入模式下，CTRL-T 命令会加入一个 'shiftwidth' 宽度的缩进，无论光标在当前行的什么位置。

你继续输入第二行，按下回车后又输入了第三行。现在的缩进一切正常。然后你按下回车输入最后一行，现在的情况如下所示：

```
if test -n a; then
   echo a
   echo "-------"
   fi
```

要删除这个多余的缩进，可以在最后一行输入 CTRL-D。这会删除一个 'shiftwidth' 宽度的缩进，无论光标在行中的什么位置。

在普通模式下，你可以用 ">>" 和 "<<" 命令来完成缩进的修正。'>' 和 '<' 是操作符，因此你可以使用通常的那三种方式来指定你要缩进的行。一个有用的组合是：

```
>i{
```

这个命令会缩进当前 {} 区内的行，'{' 和 '}' 本身并不被缩进。">a{" 会包括它们。在下面的例子中，光标停留在 "printf" 上：

```
原文                    ">i{" 之后              ">a{" 之后

if (flag)               if (flag)               if (flag)
{                       {                           {
printf("yes");              printf("yes");          printf("yes");
flag = 0;               flag = 0;               flag = 0;
}                       }                           }
```

30.5 制表符和空格

'tabstop' 在缺省状态下被设定为 8。尽管你可以改变它，但很快你就会遇到麻烦。其它的程序不知道你用的制表符间隔值是多少，你的文件看起来会一下子改变许多。另外，很多打印机都将制表符间隔值固定为 8。所以最好还是保留 'tabstop' 值不变。(如果你编辑使用其它制表符间隔值的文件，请参考 25.3 来修正。)

如果使用 8 个空格来缩进程序，你很快就会走到窗口的最右端；而用 1 个空格又看不出足够的差别。因此很多人喜欢用 4 个空格。这的确是个很好的折衷。

由于一个制表符（<Tab>）是 8 个空格，而你又想使用 4 个空格来缩进，这样你就无法使用制表符来完成缩进。这里有两种解决办法：

1. 混合使用制表符和空格。由于一个制表符占用 8 个空格的位置，你的文件会含有更少的字节数。插入或删除一个制表符也要比 8 个空格快很多。

2. 只用空格。这就避免了那些使用不同制表符间隔值的文件所带来的麻烦。

幸运的是，Vim 能够同时很好地支持这两种方式。

混合使用空格和制表符

如果你使用制表符和空格的组合，你直接按正常情况编辑就行。Vim 缺省状态下，能够很好地处理这些情况。

通过设定 'softtabstop' 可以使工作变得更简便。这个选项能使 <Tab> 看起来像是被设定为 'softtabstop' 所指定的值，但实际上使用的确是制表符和空格的组合。

当你执行下面的命令后，你每次按下 <Tab> 键，光标都会移动到下一个 "4 列" 边界：

```
:set softtabstop=4
```

当你在第一列按下 <Tab> 键后，4 个空格会插入到文本中；再次按下 <Tab> 键，Vim 会先删除那 4 个空格，然后再插入一个制表符。Vim 会尽可能地使用制表符，并辅以空格填补。

删除会以相反的方式进行。<BS> 键总是删除 'softtabstop' 指定的数量。Vim 尽可能地使用制表符，而用空格来填补空隙。

下面的例子显示了多次输入制表符然后使用 <BS> 的情况。"." 代表一个空格而 "------->" 代表制表符。

输入	结果
<Tab>	....
<Tab><Tab>	------->
<Tab><Tab><Tab>	------->....
<Tab><Tab><Tab><BS>	------->
<Tab><Tab><Tab><BS><BS>	....

另一种方法是使用 'smarttab' 选项。当它被设定，Vim 对每个在缩进行中的制表符使用 'shiftwidth'，而对在第一个非空字符后输入的 <Tab> 使用真的制表符。但 <BS> 键不会像在 'softtabstop' 选项下那样工作。

只用空格

如果你不想在文件中出现制表符，可以设定 'expandtab' 选项：

```
:set expandtab
```

当这个选项被设定，<Tab>键会插入一系列的空格。这样你可以获得如同插入一个制表符一样数量的空格。但你的文件中并不包含真正的制表符。

退格键（<BS>）每次只能删除一个空格。这样如果你键入了一个 <Tab>，你需要键入 8 次 <BS> 才能恢复。如果你在调整缩进中，输入 CTRL-D 会更快些。

制 表 符 与 空 格 的 相 互 转 换

设定 'expandtab' 选项并不会影响已有的制表符。如果你想将制表符转换为空格，可以用 ":retab" 命令。使用下面的命令：

```
:set expandtab
:%retab
```

Vim 会在所有缩进中使用空格而非制表符。但是，所有非空字符后的制表符不会受到影响。如果你想要转化这些制表符，需要在命令中加入 !：

```
:%retab!
```

这不大安全。因为它也许会修改字符串内的制表符。要检查这种情况是否存在，可以执行：

```
/"[^"\t]*\t[^"]*"
```

这里建议你不要在字符串中直接使用制表符。请用 "\t" 来替代，麻烦会少些。

将空格转化为制表符的命令则恰好相反：

```
:set noexpandtab
:%retab!
```

30.6 排版注释格式

Vim 最了不起的地方之一就是它理解注释。你可以要求 Vim 排版一段注释。它会做得很出色。

比如，你有下面的一段注释：

```
/*
 * This is a test
 * of the text formatting.
 */
```

你可以要求 Vim 排版这段注释。将光标定位到注释开头，然后输入：

```
gq]/
```

"gq" 是用来排版文本的操作符。"]/" 是移动到注释尾的动作。命令的结果是：

```
/*
 * This is a test of the text formatting.
 */
```

注意 Vim 可以正确处理每行的开头。

另外一种方法是在可视模式下用 "gq" 排版选中的文本。

要在注释中加入新的一行，先将光标移到中间一行，然后按 "o"。结果会如下所示：

```
/*
 * This is a test of the text formatting.
```

```
 *
 */
```

Vim 会为你自动添加一个星号和空格，现在你可以输入新的注释。如果一行注释长于 'textwidth'，Vim 会将其自动分开。同样，星号和空格会被自动添加进来：

```
/*
 * This is a test of the text formatting.
 * Typing a lot of text here will make Vim
 * break
 */
```

要使用这些功能，你必须在 'formatoptions' 选项中指定一些标志位：

```
r       在插入模式下，输入回车时插入星号。
o       在普通模式下，使用 "o" 或 "O" 时插入星号。
c       根据 'textwidth' 将注释分行。
```

更多标志位请参考 fo-table 。

定义注释

'comments' 选项可以定义注释的样式。Vim 可以分辨单行注释和那些包含开头，中间，结尾三部分的注释。

很多单行注释都是以一个特殊的字符开头。在 C++ 中是 //，在 Makefile 中是 #，在 Vim 脚本中是 "。比如，要使 Vim 理解 C++ 注释：

```
:set comments=://
```

冒号将条目的标志位和用来识别注释的字符分开。'comments' 的一般格式是：

```
{flags}:{text}
```

{flags} 部分可以为空（就像本例）。

不同的条目可以连接在一起，用逗号隔开。这样可以在同时识别多种不同的注释。比如，让我们编辑一个 email 信息。当回复时，别人写的内容会以 ">" 和 "!" 字符开头：

```
:set comments=n:>,n:!
```

这里有两个条目。一个识别以 ">" 开头的注释，一个识别以 "!" 开头的注释。两个都设定了 "n" 标志位，这意味着注释可以嵌套。也就是说，一个以 ">" 开始的行可以在 ">" 之后包括其它的注释符号。这样就可以用来排版下面的文本了：

```
> ! Did you see that site?
> ! It looks really great.
> I don't like it.  The
> colors are terrible.
What is the URL of that
site?
```

试着将 'textwidth' 设定为其它的值，例如 80。在可视模式下选中注释，然后输入 "gq"。结果为：

```
> ! Did you see that site? It looks really great.
> I don't like it.  The colors are terrible.
What is the URL of that site?
```

你会注意到 Vim 并没有将文本从一种注释移动到另一种注释。因为第一行以 ">!" 开头，而第二行以 ">" 开头，Vim 知道它们是不同的注释，所以第二行的 "I" 并没有移到上一行。

包括三部分的注释

C 语言注释的样式是：以 "/*" 开头，中间含有 "*"，以 "*/" 结尾。我们可以通过 'comments' 选项来这样进行设定：

```
:set comments=s1:/*,mb:*,ex:*/
```

开始部分用 "s1:/*" 定义。"s" 表示三部分注释的开始。冒号将标志位与代表注释的特殊字符 "/*" 分开。这里有一个标志位 "1"。它指明注释的中间部分有一个空格位置的偏移。

"mb:*" 的 "m" 表示这是注释的中间部分。"b" 标志位表示星号后要有空格。否则 Vim 会将形如 "*pointer" 的语句视为某个注释的中间部分。

"ex:*/" 中的 "e" 表示注释的结尾。"x" 标志位表示在 Vim 自动插入星号后，输入 "/" 会删除多余的空格。

要了解更多细节，请参考 format-comments 。

下一章: usr_31.txt 使用 GUI

VIM 用户手册 - by Bram Moolenaar
译者：lang2

利用 GUI

Vim 可以很好地在终端内工作，但是 GUI 也有其优点。比如：一个使用文件的命令可以利用文件浏览器，需要作选择时可以利用对话框。使用快捷键可以更方便的使用菜单。

后一章： usr_32.txt 撤销树
前一章： usr_30.txt 编辑程序
目录： usr_toc.txt

31.1 文件浏览器

在使用 File/Open... 菜单的时候你会看到一个文件浏览器。这会使你更容易找到要编辑的文件。但是如果你想把窗口分割并编辑另一个文件呢？没有一个菜单项是做这个的。你当然可以先用 Window/Split 然后再用 File/Open...，但这样适得其反。

大多数时候你使用 Vim 都是键入命令的，同样，打开一个文件浏览器也可以。为了让 split 命令使用文件浏览器，在它前面加上 "browse" 就行了：

```
:browse split
```

选中一个文件后 ":split" 命令就会与其一起被执行。如果你取消文件对话框就什么都不会发生。当前窗口也不会被分割。

你也可以指定一个文件名作为参数。这将告诉浏览器从那里开始查找文件。例如：

```
:browse split /etc
```

文件浏览器将以 "/etc" 作为起始目录。

":browse" 命令几乎可被添加在任何打开文件的命令前。

如果没有指定目录，Vim 会自己决定从哪里开始浏览。缺省情况 Vim 会从上次同样的目录开始。这样当你用 ":browse split" 命令并选中了一个 "/usr/local/share" 里的文件，下次你在用 ":browse" 时，浏览将从 "/usr/local/share" 开始。

用 'browsedir' 选项可以改变浏览的启动目录。可能的选项值包括：

```
last            使用上次的目录 (缺省)
buffer          使用当前缓冲区所在的目录
current         使用当前目录
```

例如，当你在 "/usr" 目录内编辑文件 "/usr/local/share/readme" 时，那么命令：

```
:set browsedir=buffer
:browse edit
```

会从 "/usr/local/share" 开始浏览，类似地：

```
:set browsedir=current
:browse edit
```

会从 "/usr" 开始浏览。

> 备注：
> 为了避免使用鼠标，多数文件浏览器提供键盘操作。这些操作因系统而异，这里不作解释。可能的情况下 Vim 使用系统的标准浏览器。请参阅你所用系统的文档。

当你不是使用 GUI 版本时，你也可以使用文件浏览窗口来选择文件。然而，":browse" 命令就无效了。参阅 netrw-browse 。

31.2 确认

Vim 会保护你的文件不被意外的覆盖或者其它的信息丢失。如果你要作一些可能有危险的事，Vim 会以一个错误信息提醒你并建议在命令后加上 ！ 来确认你希望进行该操作。

为了避免重新输入一个带有 ！ 的命令，你可以要求 Vim 用一个对话框来向你询问。你就可以选择 "OK" 或 "Cancel" 来告诉 Vim 你的要求。

例如，你正在编辑一个文件并做了一些改动。你要开始编辑另一个文件：

```
:confirm edit foo.txt
```

Vim 会弹出一个类似下面的对话框：

```
+-----------------------------------+
|                                   |
|   ?   Save changes to "bar.txt"?  |
|                                   |
|   YES   NO             CANCEL     |
+-----------------------------------+
```

你可以作选择了。如果你希望保存变动，选择 "YES"。如果你想放弃变动: "NO"。如果你想放弃编辑新文件的操作而返回来看看自己都做了那些改动用 "CANCEL"。 你会回到原来的文件，你所做的改动也都还在。

就像 ":browse" 一样，":confirm" 命令也可以被加在多数编辑其它文件的命令之前。你还可以把它们两个联起来用：

```
:confirm browse edit
```

如果当前缓冲区被改动的话这个命令会产生一个对话框。接着会弹出一个文件浏览器来选择将要编辑的文件。

> 备注：
> 在对话框中你可以使用键盘来作选择操作。通常来说 <Tab> 键和光标键可以改变选项。键入 <Enter> 会确认选项。但这也跟你用的系统有关。

当你不是使用 GUI 版本的时候，":confirm" 也是有效的。Vim 不会弹出一个对话框，而是将询问显示在 Vim 窗口的底部并提示你键入选择。

```
:confirm edit main.c
Save changes to "Untitled"?
[Y]es, (N)o, (C)ancel:
```

你可以键入单键来作出选择。不用键入 <Enter>，这和其它命令行的输入不同。

31.3 菜单快捷键

所有的 Vim 命令都是用键盘来完成的。在不知道命令名称的情况下，使用菜单会简单些。但是你就得把手从键盘上移开去抓鼠标。

通常菜单用键盘也可以操作。这决定于你所使用的系统，但多数情况下是这样工作的：将 <Alt> 键和菜单项中带下划线的字母连用。例如，<A-w> (<Alt> 加 w) 弹出 Window 菜单。

在 Window 菜单下，"split" 菜单项中的 p 下面划了线。放开 <Alt> 键然后按 p 就可以选中它了。

在用 <Alt> 键选中菜单后，你可以用光标键来在菜单内移动。<Right> 选择一个子菜单，<Left> 关闭之。<Esc> 也用来关闭菜单。<Enter> 选中一个菜单项。

使用 <Alt> 键来操作菜单和使用 <Alt> 键的映射会出现冲突。'winaltkeys' 可以用来告诉 Vim 如何对待 <Alt> 键。

缺省值 "menu" 是一个明智的选择：如果该键组合是一个菜单快捷键那么就不能被作为映射。所有其它的键都可以。

"no" 表示不使用 <Alt> 键组合来操作菜单。这样你就必须使用鼠标。所有 <Alt> 键组合都可以被用作键盘映射。

"yes" 表示 Vim 会使用 <Alt> 键组合来操作菜单。另外的 <Alt> 键组合也可以用作其它用途。

31.4 Vim 窗口位置及大小

要查看当前 Vim 窗口在屏幕上的位置可以用：

```
:winpos
```

这只对 GUI 有效。输出可能是这样的：

```
Window position: X 272, Y 103
```

位置是以屏幕像素为单位的。你可以通过数字来将 Vim 窗口移动到别处。例如，将其向左移动一百个像素：

```
:winpos 172 103
```

备注：
报告的窗口位置和窗口被移动的位置可能会有小的出入。这是由窗口周围的边框引起的。边框是被窗口管理器加上的。

你可以在你的启动脚本中使用这个命令来将窗口定位到一个指定的位置。

Vim 窗口的大小是以字符数目计算的。因此它和所使用的字体的大小有关。你可以查看当前的窗口大小:

```
:set lines columns
```

要改变窗口大小只要改变 'lines' 和/或 'columns' 选项的值即可:

```
:set lines=50
:set columns=80
```

取得窗口大小的操作在终端和在 GUI 上都可以。但是设定大小在绝大多数终端里都是不可能的。

在启动 X-Window 版本的 gvim 时你可以在命令行指定窗口的位置和大小:

```
gvim -geometry {width}x{height}+{x-offset}+{y-offset}
```

{width} 和 {height} 的单位是字符。{x-offset} 和 {y-offset} 的单位是像素。例:

```
gvim -geometry 80x25+100+300
```

31.5 杂项

你可以使用 gvim 来编辑一个 e-mail。在你的 e-mail 程序里你得选择 gvim 作为其编辑程序。当你尝试这种配置的时候，你会发现行不通: 邮件程序认为编辑已经结束了，可实际上 gvim 还在运行着!

这里的情况是由于 gvim 启动时会将自己从外壳分离出来的缘故。如果你是从终端启动 gvim 的话这当然没有问题，你还可以在外壳里作其它事。但如果你的确希望等待 gvim 结束的话，你必须阻止这个分离动作。"-f" 参数就是用来做这个的:

```
gvim -f file.txt
```

"-f" 代表前台 (foreground)。这样 Vim 就会挂起它启动所在的外壳，直到你编辑完毕并退出为止。

推 迟 启 动 GUI

在 Unix 上你可以先在终端里启动 Vim。当你在一个外壳里完成各种不同任务的时候这很有方便。如果当你在编辑某个文件时决定你想使用 GUI，你可以这样做:

```
:gui
```

Vim 就会打开 GUI 窗口并不再使用终端。你可以继续在终端里作其它事情。"-f" 参数在这里也可以用来将 GUI 放置在前台: ":gui -f"。

GVIM 启 动 文 件

当 gvim 启动时，它会读取 gvimrc 文件。这和启动 Vim 时用到的 vimrc 文件类似。gvimrc 文件可以被用来为 GUI 设定专用的选项和命令。例如，你可以设定 'lines' 的值来指定窗口的大小:

```
:set lines=55
```

在终端里使用是没有用的，因为它的大小是固定的（除了那些支持调整大小的 xterm）。
Vim 在 vimrc 文件相同的地方找寻 gvimrc 文件。通常来说在 Unix 上是 "~/.gvimrc"； MS-Windows 上是 "$VIM/_gvimrc"。$MYGVIMRC 环境变量设为该值，这样你可以用下面的命令编辑此文件，如果有的话：

```
:edit $MYGVIMRC
```

如果出于某种原因你不想使用通常的 gvimrc 文件，你可以利用 "-U" 参数来指定另外的一个：

```
gvim -U thisrc ...
```

这使你可以用 gvim 来做另外一类编辑。例如你可以用另外的字体大小。
要完全跳过读取 gvimrc 文件：

```
gvim -U NONE ...
```

下一章： usr_32.txt 撤销树

usr_32.txt　　适用于 Vim 9.1 版本。　　最近更新：2010年7月

VIM 用户手册 - by Bram Moolenaar
译者：Willis

撤销树

Vim 提供了多层撤销功能。如果你撤销了一些改变然后又进行了一些新的改变，你就在撤销树里建立了一个分支。本文本讨论如何在分支间来回移动。

后一章：　usr_40.txt　　创建新的命令
前一章：　usr_31.txt　　利用 GUI
目录：　usr_toc.txt

32.1　撤销到文件写入时的状态

有时你做了一些改变，然后发现还是想恢复到最近写入文件时的状态。没问题，用下面的命令就可以：

```
:earlier 1f
```

这里，"f" 代表 "file"（文件）。

可以重复此命令，回到更遥远的过去。使用不同于 1 的计数值可以回去得快一些。

如果回去太久了，可以这样往前：

```
:later 1f
```

注意　这些命令真的是依据时间序列进行的。如果你在撤销一些改变后又进行过改变，这一点很有区别。下一节有所解释。

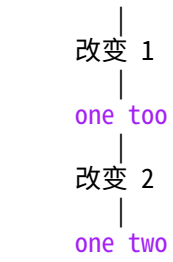
另外也请　注意　我们说的是文本写入。要把撤销信息写入文件见　undo-persistence　。

32.2　为每次改变进行编号

02.5　一节，我们只讨论了单线的撤销/重做。其实，可以出现分支。当你撤销了一些改变，又进行了一些新的改变，新的改变构成了撤销树的一个分支。

让我们从文本 "one" 开始。第一个要做的改变是附加 " too"。然后移动到第一个 'o' 上并修改为 'w'。这时我们有了两个改变，分别编号为 1 和 2，而文本有三个状态：

```
one
```



如果我们撤销一次改变，回到 "one too"，然后把 "one" 换成 "me"，我们就在撤销树里建立了一个分支：

现在你可以用 u 命令来撤销。如果你做两次，你得到的是 "one"。用 CTRL-R 来重做，你会到达 "one too"。多做一次 CTRL-R 又把你带到 "me too"。现在我们看到，撤销/重做使用最近使用的分支，在树内上下移动。

这里重要的是改变发生的顺序。这里说的改变不考虑撤销和重做。每次改变后，你会得到一个新的文本状态。

注意 只有改变被编号，上面显示的文本没有标识符。通常，通过它上方的改变号来引用它。但有时也通过他下方的某个改变之一来引用。特别是在树内往上移动的时候，这样你可以知道哪个改变刚刚被撤销掉。

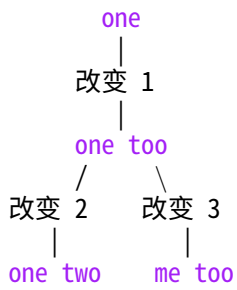
32.3 撤销树内任意跳转

那么你现在怎么能到 "one two" 呢？你可以用这个命令：

```
:undo 2
```

文本现在成为了 "one two"，因为它在改变号 2 之下。用 :undo 命令可以跳转到树内任何改变之下的文本。

现在再进行一项改变：把 "one" 改成 "not"：



```
    |
改变 4
    |
not two
```

现在你又改了主意想回到 "me too" 了。用 g- 命令。它在时间点上往后退，也就是说，它不是在树内上下移动，而是回到之前所在的改变。

你可以重复 g- ，这样你会看到文本的变化过程：

```
me too
one two
one too
one
```

用 g+ 时间点上往前进：

```
one
one too
one two
me too
not two
```

:undo 用于你知道你要跳转到哪个改变的场合。 g- 和 g+ 用于你不知道具体要到达的改变号的情况。

你可以在 g- 和 g+ 之前加上计数来重复执行。

32.4 时间旅行

如果你在文本上工作了一段时间，撤销树变得相当大。这时你可能想回到几分钟之前的文本上。

要看撤销树里有什么分支，用：

```
:undolist
number changes  time
     3       2  16 seconds ago
     4       3  5 seconds ago
```

这里你可以看到每个分支上叶结点的编号，还有改变发生的时间。假定我们在改变号 4 下方的 "not two" 那里，你可以这样回到十秒前：

```
:earlier 10s
```

取决于改变发生了多长时间，你回到达树内的某个位置。 :earlier 命令参数可以用 "m" 代表分钟，"h" 代表小时，"d" 代表天。用一个很大的数，你可以一路回到很久以前：

```
:earlier 100d
```

要（再次）进入未来世界，用 :later 命令：

```
:later 1m
```

参数可以用 "s"、"m" 和 "h"，和 :earlier 完全类似。

要看到更多的细节，或者要对信息进行操作，可以用 undotree() 函数。要看看它返回什么：

```
:echo undotree()
```

下一章： usr_40.txt 创建新的命令

VIM 用户手册 - by Bram Moolenaar
译者：lang2

创建新的命令

Vim 是一个可扩展的编辑器。你可以把一系列你常用的命令组合成一个新的命令。或者重新定义一个现存的命令。各种命令的自动执行可以通过自动命令实现。

40.1 键映射

简单的映射已经在 05.4 介绍过了。基本的概念是将一系列的键输入转换成为另外一个键输入序列。这是一个很简单，但是很有效的机制。

最简单的形式是将一个键输入序列映射到一个键上。由于那些除了 <F1> 外的功能键都没有预先定义的功能，选择它们作为映射对象是很有效的。例如：

```
:map <F2> GoDate: <Esc>:read !date<CR>kJ
```

这显示了如何使用三种不同的运行模式。在用 "G" 移动到最后一行后，"o" 命令开始一个新行并开始插入模式。然后文本 "Date: " 被输入并用 <Esc> 离开插入模式。

注意 在 <> 内使用的特殊键。这叫尖括号记法。你要分别地输入这些字符，而不是键入要表示的键本身。这使得映射更具可读性，而且你也可以方便地拷贝，粘贴文本。

":" 使得 Vim 回到命令行。":read !date" 命令读取 "date" 命令的输出并添加到当前行之下。<CR> 是用来执行该命令的。

到此为止，文本看起来像：

```
Date:
Fri Jun 15 12:54:34 CEST 2001
```

然后 "kJ" 将光标上移并将两行连接起来。

参阅 map-which-keys 可以帮助你决定应该使用哪些键来作映射。

映 射 与 运 行 模 式

":map" 命令定义普通模式的键映射。你也可以为其它运行模式定义映射。例如，":imap" 用来定义插入模式的映射。你可以用它来定义一个插入日期的映射：

```
:imap <F2> <CR>Date: <Esc>:read !date<CR>kJ
```

看起来很像前面为普通模式定义的 <F2> 映射。只是开始的地方有所不同。普通模式下的

<F2> 映射依然有效。这样你就可以在各种模式下为同一映射键定义不同的映射。

应该注意的是，虽然这个映射以插入模式开始，但它却以普通模式结束。如果你希望继续插入模式，可以在最后加上 "a"。

下面是一个映射命令及其生效模式的总览：

```
:map            普通，可视模式及操作符等待模式
:vmap           可视模式
:nmap           普通模式
:omap           操作符等待模式
:map!           插入和命令行模式
:imap           插入模式
:cmap           命令行模式
```

操作符等待模式是当你键入一个操作符（比如 "d" 或 "y"）之后，Vim 期待你键入一个动作命令或者文本对象时的状态。比如，当你键入命令 "dw"， 那个 "w" 就是在操作符等待模式下键入的。

假定你想定义映射 <F7> 使得命令 d<F7> 删除一个 C 程序块（{} 包括的文本）。类似的 y<F7> 会将程序块拷贝到匿名的寄存器。因此，你所要做的就是定义 <F7> 来选择当前的语法块。你可以用下面的命令做到：

```
:omap <F7> a{
```

这使得 <F7> 在操作符等待模式下选择一个块，就像是你键入了 "a{" 一样。这个映射在你不容易键入 { 时比较有用。

映 射 列 表

要查看当前定义的映射，使用不带参数的 ":map" 命令。或者其它运行模式的变体。输出应该类似于：

```
   _g            :call MyGrep(1)<CR>
v  <F2>          :s/^/> /<CR>:noh<CR>``
n  <F2>          :.,$s/^/> /<CR>:noh<CR>``
   <xHome>       <Home>
   <xEnd>        <End>
```

第一列显示该映射有效的运行模式。"n" 表示普通模式，"i" 表示插入模式，等等。空白表示用 ":map" 命令定义的映射，也就是对普通和可视模式有效。

列出映射的一个比较实用的目的是检查 <> 表示的特殊键是否被识别了（仅当支持多色彩是有效）。例如，当 <Esc> 被用彩色显示时，它表示转义字符。否则，只是 5 个不同的字符。

重 映 射

映射的结果会检查其中包括的其他映射。例如，上面对 <F2> 的映射可以减短为：

```
:map <F2> G<F3>
:imap <F2> <Esc><F3>
:map <F3>  oDate: <Esc>:read !date<CR>kJ
```

在普通模式下 <F2> 被映射为：行进至最后一行，然后输入 <F3>；在插入模式下先键入 <Esc> 后也输入 <F3>。接下来 <F3> 也被映射，执行真正的工作。

假设你几乎不使用 Ex 模式，并想用 "Q" 命令来排版文本（就像旧版本的 Vim 那样）。下面的映射就能做到：

```
:map Q gq
```

但是，你总有需要用到 Ex 模式的时候。我们来将 "gQ" 映射为 Q，这样你仍然可以进入 Ex 模式：

```
:map gQ Q
```

这样一来当你键入 "gQ" 时它被映射为 "Q"。到现在为止一切顺利。但由于 "Q" 被映射为 "gq"， 输入的 "gQ" 被解释成为 "gq"， 你根本就没进入 Ex 模式。

要避免键被再次映射，使用 ":noremap" 命令：

```
:noremap gQ Q
```

现在 Vim 就知道了对 "Q" 不需要检查与之相关的映射。对于每个模式都有一个类似的命令：

```
:noremap        普通，可视和操作符等待模式
:vnoremap       可视模式
:nnoremap       普通模式
:onoremap       操作符等待模式
:noremap!       插入和命令行模式
:inoremap       插入模式
:cnoremap       命令行模式
```

递 归 映 射

当一个映射调用它本身的时候，会无限制的运行下去。这可以被用来无限次重复一个操作。

例如，你有一组文件，每个的第一行都包括一个版本号。你用 "vim *.txt" 来编辑它们。你现在正在编辑第一个文件。定义下面的映射：

```
:map ,, :s/5.1/5.2/<CR>:wnext<CR>,,
```

现在当你键入 ",," 时，上面的映射被触发。它把第一行的 "5.1" 替换为 "5.2"。接着执行 ":wnext" 来写入文件并开始编辑下一个。映射以 ",," 结束。这又触发了同一个映射，再次执行替换操作，依此类推。

这个映射会一直进行下去，直至遇到错误为止。在这里可能是查找命令无法匹配到 "5.1" 。你可以自行插入 "5.1" 然后再次键入 ",,"。或者 ":wnext" 因为遇到最后一个文件而失败。

当映射在中途遇到错误时，映射的剩余部分会被放弃。你可用 CTRL-C 中断映射。(在 MS-Windows 上用 CTRL-Break)。

删 除 映 射

要删除一个映射，使用 ":unmap" 命令。同样，删除映射的命令也和运行模式相关：

:unmap	普通，可视和操作符等待模式
:vunmap	可视模式
:nunmap	普通模式
:ounmap	操作符等待模式
:unmap!	插入和命令行模式
:iunmap	插入模式
:cunmap	命令行模式

这里有个小技巧可以定义一个对普通模式和操作符等待模式有效而对可视模式无效的映射：先对三个模式都定义映射，然后将可视模式的那个删除：

```
:map <C-A> /--><CR>
:vunmap <C-A>
```

注意那 5 个字符 "<C-A>" 表示一个键组合 CTRL-A。

要清除所有的映射，使用 :mapclear 命令。现在你应该可以猜到各种模式下的变体了吧。要当心使用这个命令，它不可能被撤销。

特 殊 字 符

在 ":map" 命令后面可以追加另一个命令。需要用 | 字符来将两个命令分开。这也就意味着一个映射中不能直接使用该字符。需要时，可以用 <Bar> (五个字符)。例如：

```
:map <F8> :write <Bar> !checkin %:S<CR>
```

":unmap" 命令有同样的问题，而且你得留意后缀的空白字符。下面两个命令是不同的：

```
:unmap a | unmap b
:unmap a| unmap b
```

第一个命令试图删除映射 "a "，后面带有一个空格。

当要在一个映射内使用空格时，应该用 <Space> (七个字符)：

```
:map <Space> W
```

这使得空格键移动到下一个空白字符分割的单词。

在一个映射后不能直接加注释，因为 " 字符也被当作是映射的一部分。你可以用 |" 绕过这一限制。这实际上是开始一个新的空命令。例如：

```
:map <Space> W|     " Use spacebar to move forward a word
```

映 射 与 缩 写

缩写和插入模式的映射很像。对参数的处理它们是一样的。它们主要的不同在于触发的方式。缩写是由单词之后的非单词字符触发的。而映射由其最后一个字符触发。

另一个区别是你键入的缩写的字符会在你键入的同时被插入到文本内。当缩写被触发时，这些字符会被删除并替换成缩写所对应的字符。当你键入一个映射时，直到你完成所有的映射键而映射被触发时，映射所对应的内容才会被插入。如果你置位 'showcmd' 选

项，键入的字符会显示在 Vim 窗口的最后一行。
有一个例外是当映射有歧义的时候。假定你有两个映射:

```
:imap aa foo
:imap aaa bar
```

现在，当你键入 "aa" 时，Vim 不知道是否要使用第一个映射。它会等待另一个键输入。如果是 "a"，第二个映射被执行，结果是 "bar"。如果是其它字符，例如空格，第一个映射被执行，结果是 "foo"， 而且空格字符也会被插入。

另 外 ...

<script> 关键字可以被用来使一个映射仅对当前脚本有效。参见 :map-<script> 。

<buffer> 关键字可以被用来使一个映射仅对当前缓冲区有效。参见 :map-<buffer> 。

<unique> 关键字可以被用来当一个映射已经存在时不允许重新定义。否则的话新的映射会简单的覆盖旧的。参见 :map-<unique> 。

如果要使一个键无效，将之映射至 <Nop> (五个字符)。下面的映射会使 <F7> 什么也干不了:

```
:map <F7> <Nop>| map! <F7> <Nop>
```

注意 <Nop> 之后一定不能有空格。

40.2 定义命令行命令

Vim 编辑器允许你定义你自己的命令。你可以像运行其他命令行命令一样运行你自定义的命令。
要定义一个命令，像下面一样执行 ":command" 命令:

```
:command DeleteFirst 1delete
```

现在当你执行 ":DeleteFirst" 命令时，Vim 执行 ":1delete" 来删除第一行。

> 备注:
> 用户定义的命令必须以大写字母开始，但不能用 ":X"，":Next" 和 ":Print"。也不能用下划线! 你可以使用数字，但是不鼓励这么做。

要列出用户定义的命令，执行下面的命令:

```
:command
```

像那些内建的命令一样，用户自定义的命令也可以被缩写。你只需要键入足够区别于其它命令的字符就可以了。命令行补全也有效。

参 数 个 数

自定义命令可以带一系列的参数。参数的数目必须用 -nargs 选项来指定。例如，上面例子中的 :DeleteFirst 命令不带参数，所以你也可以这样来定义:

```
:command -nargs=0 DeleteFirst 1delete
```

不过，因为缺省参数数目为 0，你没有必要加上 "-nargs=0"。其它可用的值是：

```
-nargs=0        无参数
-nargs=1        一个参数
-nargs=*        任意数目的参数
-nargs=?        没有或一个参数
-nargs=+        一个或更多参数
```

使 用 参 数

在命令的定义中，<args> 关键字可以用来表示命令带的参数。例如：

```
:command -nargs=+ Say :echo "<args>"
```

现在当你输入

```
:Say Hello World
```

Vim 会显示 "Hello World"。然而如果你加上一个双引号，就不行了。例如：

```
:Say he said "hello"
```

要把特殊字符放到字符串里，必须在它们的前面加上反斜杠，用 "<q-args>" 就可以：

```
:command -nargs=+ Say :echo <q-args>
```

现在上面的 ":Say" 命令会引发下面的命令被执行：

```
:echo "he said \"hello\""
```

关键字 <f-args> 包括与 <args> 一样的信息，不过它将其转换成适用于函数调用的格式。例如：

```
:command -nargs=* DoIt :call AFunction(<f-args>)
:DoIt a b c
```

会执行下面的命令：

```
:call AFunction("a", "b", "c")
```

行 范 围

有些命令需要一个范围作为参数。要告诉 Vim 你需要定义这样的命令，使用 -range 选项。它可能的值如下：

```
-range          允许范围；缺省为当前行。
-range=%        允许范围；缺省为整个文件。
-range={count}  允许范围；只用该范围最后的行号作为单个数字的参数，其缺
                省值为 {count}。
```

当一个范围被指定时，关键字 <line1> 和 <line2> 可以用来取得范围的首行和末行的行号。例如，下面的命令定义将指定的范围写入文件 "save_file" 的命令 - SaveIt:

```
:command -range=% SaveIt :<line1>,<line2>write! save_file
```

其 它 选 项

其它的一些选项有:

-count={number}	命令可以带 count 参数，缺省为 {number}。用 <count> 关键字可以访问该参数。
-bang	允许使用 !。若 ! 出现，<bang> 扩展为 !。
-register	你可以指定一个寄存器。(缺省为无名寄存器。)指定的寄存器可通过 <reg> (即 <register>) 来操作。
-complete={type}	给出命令行补全的方式。 :command-completion 列出了所有可用值。
-bar	命令后可用 | 加另外一个命令，或 " 加一个注释。
-buffer	命令仅对当前缓冲区有效。

最后，你还可以使用 <lt> 关键字来代表字符 <。这样可以转义上面提到的 <> 项目的特殊含义。

重 定 义 和 删 除

! 参数可以用来重新定义相同的命令:

```
:command -nargs=+ Say :echo "<args>"
:command! -nargs=+ Say :echo <q-args>
```

要删除自定义命令，使用 ":delcommand"。该命令只带一个参数，那就是自定义命令的名字。例:

```
:delcommand SaveIt
```

要一次删除所有的自定义命令:

```
:comclear
```

要当心! 这个命令无法撤销。

关于所有这些内容的更多信息可参阅参考手册： user-commands 。

40.3 自动命令

自动命令是一类特殊的命令。当某些事件，例如文件读入或改变缓冲区等事件发生时，它们会自动被执行。例如，通过自动命令你可以教 Vim 来编辑压缩文件。这个功能被用在 gzip 插件里。

自动命令非常强大。如果你小心使用的话，自动命令可以省去你很多自己敲命令的麻烦。如果不当心的话你就是自找麻烦。

假设你希望在每次写入文件时自动的替换文件尾部的日期戳。先定义一个函数：

```
:function DateInsert()
:  $delete
:  read !date
:endfunction
```

你需要在每次缓冲区写入文件之前想办法调用该函数。下面这一行就能做到：

```
:autocmd BufWritePre *  call DateInsert()
```

"BufWritePre" 是这个自动命令的触发事件：把缓冲区写入文件前（pre）。"*" 是一个用来匹配文件名的模式。这儿它匹配所有文件。

如果这个命令生效，当你调用 ":write" 时，Vim 检查是否有匹配 BufWritePre 事件的自动命令并执行它们。然后才执行 ":write"。

通用的 :autocmd 命令格式如下：

```
:autocmd [group] {events} {file-pattern} [++nested] {command}
```

组名 [group] 是可选的。它被用来管理和调用命令（后面再讲）。{events} 参数是一个触发事件列表（用逗号隔开）。

{file-pattern} 是文件命令，通常带有通配符。例如，用 "*.txt" 会使得自动命令对所有文件名以 ".txt" 结尾的文件被调用。可选项 [++nested] 允许自动命令的嵌套（见下）。最后，{command} 是要被执行的命令。

增加自动命令时，原有的自动命令依然保留。要避免多次加入自动命令可用以下形式：

```
:augroup updateDate
:  autocmd!
:  autocmd BufWritePre *  call DateInsert()
:augroup END
```

定义新的自动命令之前， :autocmd! 先删除之前定义的自动命令。关于组以后细讲。

事 件

最有用的事件之一是 BufReadPost。它在一个文件被调入编辑之后被触发。常被用来设定相关的选项。例如，你已知 ".gsm" 文件是 GNU 汇编程序源码。为确保使用正确的语法文件，可以定义这样的自动命令：

```
:autocmd BufReadPost *.gsm  set filetype=asm
```

如果 Vim 能够正确的识别文件类型的话，它将为你设定 'filetype' 选项。这会触发 Filetype 事件。你可以利用这个来为某一类型的文件做编辑的准备工作。例如，要为文本文件调入一组缩写：

```
:autocmd Filetype text  source ~/.vim/abbrevs.vim
```

在开始编辑一个新文件时，你可以要求 Vim 插入一个模板：

```
:autocmd BufNewFile *.[ch]  0read ~/skeletons/skel.c
```

在 autocmd-events 可以找到一个完整的事件列表。

匹配模式

那个 {file-pattern} 参数实际上可以是一个以逗号分割开的模式列表。例如: "*.c,*.h" 匹配所有文件名以 ".c" 和 ".h" 结尾的文件。
常见的文件通配符都可以使用。这里给出一个最常用的清单:

*	匹配任意字符，任意多次
?	匹配任意字符，一次
[abc]	匹配 a、b 或 c
.	匹配一个点 .
a{b,c}	匹配 "ab" 和 "ac"

当模式包括斜杠 (/) 时 Vim 会比较路径名。否则只有文件名的最后部分才用来作比较。例如，"*.txt" 匹配 "/home/biep/readme.txt"。模式 "/home/biep/*" 也可以匹配那个文件。但是 "home/foo/*.txt" 就不行。
当包括斜杠时，Vim 会试着匹配文件的完整路径 ("/home/biep/readme.txt") 和相对路径 (例如: "biep/readme.txt")。

> 备注:
> 当在使用反斜杠作为文件分隔符的系统 (如 MS-Windows) 上工作时，你也得在自动命令中使用正斜杠。这会使编写匹配模式变得容易些，因为反斜杠有特殊的意义。它同时也使自动命令更具可移植性。

删除

要删除一个自动命令，使用和定义它一样的命令格式。但不要包括后面的 {command} 部分，而且要加上 !。例如:

```
:autocmd! FileWritePre *
```

这样会删除为 "FileWritePre" 事件定义的匹配 "*" 文件名模式的所有自动命令。

列表

要列出当前定义的所有自动命令，用这个:

```
:autocmd
```

这个列表可能会相当长，特别是在使用了文件类型检测时。你可以指定组，事件和/或文件名模式来要求仅列出相关的命令。例如，要列出 BufNewFile 事件的所有自动命令:

```
:autocmd BufNewFile
```

列出所有匹配文件名模式 "*.c" 的命令:

```
:autocmd * *.c
```

使用 "*" 作为事件会给出所有事件的列表。要列出 cprograms 组对应的自动命令:

```
:autocmd cprograms
```

组

当定义自动命令时用到 {group} 这一项时，自动命令会被分成组。比如说，这可以被用来删除一个组中的所有命令。
　在为某一个组定义数个自动命令时，可以使用 ":augroup" 命令。例如，我们来定义一些用于 C 程序的自动命令：

```
:augroup cprograms
:  autocmd BufReadPost *.c,*.h :set sw=4 sts=4
:  autocmd BufReadPost *.cpp   :set sw=3 sts=3
:augroup END
```

这和下面的命令有一样的效果：

```
:autocmd cprograms BufReadPost *.c,*.h :set sw=4 sts=4
:autocmd cprograms BufReadPost *.cpp   :set sw=3 sts=3
```

要删除 "cprograms" 组中的所有自动命令：

```
:autocmd! cprograms
```

嵌 套

一般的，某一事件触发的自动命令在被执行时不会再触发其它事件。例如，当因 FileChangedShell 事件而读入一个文件时，那些被定义来设定语法的自动命令就不会被触发。要使那些命令被触发，加上一个 "nested" 参数：

```
:autocmd FileChangedShell * ++nested  edit
```

执 行 自 动 命 令

Vim 允许你用模拟某一事件发生的办法来触发一个自动命令。这可以在一个自动命令里用来触发另外一个。例如：

```
:autocmd BufReadPost *.new  execute "doautocmd BufReadPost " . expand("<afile>:r")
```

这定义了一个新文件开始编辑之后触发的自动命令。这个文件的文件名必须以 ".new" 结尾。其中的 ":execute" 命令利用表达式求值来组成一个新的命令并执行之。当编辑文件 "tryout.c.new" 时被执行的命令将是：

```
:doautocmd BufReadPost tryout.c
```

expand() 函数的参数是 "<afile>"，用来代表自动命令执行所关联的文件。":r" 指定仅使用其根部分。

":doautocmd" 执行于当前缓冲区。":doautoall" 命令于 "doautocmd" 命令类似但执行于所有缓冲区。

使 用 普 通 模 式 命 令

自动命令所执行的命令是 "命令行" 命令。如果你想在其中执行普通模式命令，可以使用 ":normal" 命令。例如:

```
:autocmd BufReadPost *.log normal G
```

这样，当你编辑 *.log 文件时 Vim 会将光标移动到最后一行。
使用 ":normal" 命令需要点技巧。首先，你要确保其参数是一个包括所有参数的完整命令。当你用 "i" 进入插入模式时，你必须用 <Esc> 离开。如果你用 "/" 来开始查找，你也必须用 <CR> 执行该查找命令。
":normal" 命令会使用其后的所有文本作为将要执行的命令。因此不可能用 | 来后跟另一个命令。有个办法可以绕过这个约束: 把 ":normal" 命令放在 ":execute" 命令之内。这个方法同时也方便了不可显示的字符作为参数的传递。例如:

```
:autocmd BufReadPost *.chg execute "normal ONew entry:\<Esc>" |
        \ 1read !date
```

上面的例子还展示了如何用反斜杠来将一个长命令分为几行。这可以用在 Vim 脚本中(不能用在命令行)。

如果你想让你的自动命令作一些复杂的操作，其中涉及在文件间跳转然后回到原来位置，你希望能够恢复文件的视窗位置。 restore-position 有些例子。

忽 略 事 件

有些时候，你并不想触发自动命令。'eventignore' 选项包括了一组会被 Vim 完全忽略的事件。例如，下面的命令会使得进入和离开窗口的事件被忽略掉:

```
:set eventignore=WinEnter,WinLeave
```

要忽略所有的事件，用下面的命令:

```
:set eventignore=all
```

要恢复到正常的状态，把 'eventignore' 设定为空即可:

```
:set eventignore=
```

下一章: usr_41.txt 编写 Vim 脚本

VIM 用户手册 - by Bram Moolenaar
译者：lang2、Willis

编写 Vim 脚本

Vim 脚本语言在很多地方用到，包括 vimrc 文件，语法文件，等等。本章讨论 Vim 脚本相关的知识。这样的内容有很多，所以本章也比较长。

下一章： usr_42.txt 添加新的菜单
前一章： usr_40.txt 创建新的命令
目录： usr_toc.txt

41.1 简介 vim-script-intro script

先从一些术语的命名说起。Vim 脚本是 Vim 可以解释和执行的文件。这包括用 Vim 的脚本语言书写的文本，如 .vim 文件或 .vimrc 和 .gvimrc 这种配置文件。这些脚本可以定义函数、命令和设置，Vim 用来定制和扩展其行为。

稍稍滥用一下这个术语，在本文档里我们将用 "Vim 脚本" 来称呼 Vim 脚本语言。这种简写可以精简一下关于 Vim 脚本编程的解释和讨论。

Vim 插件是一个或多个 Vim 脚本的集合，随带着附加文件，如帮助文档，配置文件还有其他资源，设计用来为 Vim 增加特定的特性和功能。插件可以提供新命令，增强已有的能力，还有集成外部工具或服务进 Vim 环境。

你最初接触到 Vim 脚本是在 vimrc 文件里。当 Vim 启动时它将读取该文件的内容并执行其中的命令。你可以在其中设置选项，定义映射，选择插件还有很多别的。你也可以在其中使用任何冒号命令（以 ":" 开头的命令；这些命令有时也被称作 Ex 命令或命令行命令）。

语法文件其实也是 Vim 脚本。专为某种文件类型设定选项的文件也是。一个很复杂的宏可以被单独的定义在一个 Vim 脚本文件中。你可以自己想到其它的应用。

Vim 脚本有两种风格：老式和 Vim9 。因为此帮助文件是为新手准备的，我们会教你更新更方便的 Vim9 语法。老式脚本是特别为 Vim 设计的，而 Vim9 脚本更像其它语

言，如 JavaScript 和 TypeScript。

要试试 Vim 脚本的最好办法是编辑一个脚本文件然后执行。基本上：

```
:edit test.vim
[插入想要的脚本行]
:w
:source %
```

让我们从一个简单的例子开始：

```
vim9script
var i = 1
while i < 5
  echo "count is" i
  i += 1
endwhile
```

本例的输出是：

```
count is 1
count is 2
count is 3
count is 4
```

第一行的 "vim9script" 命令明确了这是新的 Vim9 脚本文件。这对控制文件其余部分如何使用很重要。推荐放在头一行，包括在任何注释之前。

vim9-declarations

"var i = 1" 命令声明并初始化 "i" 变量。通常的用法是：

```
var {变量} = {表达式}
```

在例子中变量名是 "i" 而表达式是一个简单的数值 1。

":while" 命令开始一个循环。通常的用法是：

```
while {条件}
  {语句}
endwhile
```

只要条件为真，while 和 endwhile 包围的语句就会被执行。在例子中使用的条件是表达式 "i < 5"。这个条件在变量 i 小于五时总是真的。

备注：
如果你碰巧写了一个死循环语句，你可以用 CTRL-C 来终止（在 MS-Windows 上使用 CTRL-Break）。

echo 命令显示它的参数。在这个例子中的参数是字符串 "count is" 和变量 i 的值。因为开始时 i 的值是 1，所以将会显示：

```
count is 1
```

接着是 `i += 1` 命令。该命令相当于 "i = i + 1"。在变量 i 上加一并将新的值赋给同一个变量。

给出本例是为了解释命令，不过如果你真的要写这样一个循环，下面的表达更加简洁：

```
for i in range(1, 4)
  echo $"count is {i}"
endfor
```

我们现在不解释 for 、 range() 和 $"string" 如何工作，一会儿再说。如果你没有耐心，点击这些链接。

试 试 例 子

可以直接试试这些帮助文件里的绝大数例子，不需要先把命令保存到文件里。比如，要试试上面的 "for" 循环例子，可以这么做：
1. 把光标放在 "for" 上
2. 用 "v" 启动可视模式
3. 往下移到 "endfor"
4. 按冒号和 "so"，然后回车

按了冒号后会见到 ":'<,'>"，这代表了可视选择文本的范围。

有些命令要确保在 Vim9 脚本里执行。键入命令通常使用老式脚本语法，比如下面导致 E1004 错误的例子。要用 Vim9 语法，需要改改第四步：
4. 按冒号和 "vim9 so"，然后回车

"vim9" 是 vim9cmd 的缩写，使后续的那个命令使用 Vim9 语法的命令修饰符。

注意 这套方法不适用于需要脚本上下文的示例。

四 种 数 值

数值可以是十进制，十六进制，八进制或者二进制的。

以 "0x" 或 "0X" 开始的数值是十六进制的。例如 "0x1f" 代表十进制 31，而 "0x1234" 是十进制 4660。

以 "0o"、"0O" 开始的数值是八进制的。"017" 代表十进制 15。

以 "0b" 或 "0B" 开始的数值是二进制的。例如 "0b101" 代表十进制 5。

十进制就是由简单数位组成的数值。当心：在老式脚本里不要在十进制数前添上零，那样该数值将会被作为八进制数对待！这是建议使用 Vim9 脚本的一个原因。

echo 命令计算参数，如果是数值则总以十进制格式显示数值。例：

```
:echo 0x7f 0o36
127 30
```

在一个数值前加上负号会将其变为负值。十六进制数、八进制数和二进制数亦然：

```
echo -0x7f
-127
```

减号也用于减法操作。有时这会引起混淆。如果在两个数值前都放上负号，会报错：

```
echo -0x7f -0o36
E1004: White space required before and after '-' at "-0o36"
```

注意：如果不是在 Vim9 脚本里试这些命令而直接键入，假定使用老式脚本语法。因而 echo 命令会把第二个负号看成是减法运算。要重现此错，在命令前加上 vim9cmd ：

```
vim9cmd echo -0x7f -0o36
E1004: White space required before and after '-' at "-0o36"
```

表达式中常常需要空白字符以增加表达式的易读性和避免错误。比如你会觉得上面的 "-0o36" 会使数值变负，而实际上它被视为减法。

如果真要使减号用作数值取负，用括号包围第二个表达式：

```
echo -0x7f (-0o36)
-127 -30
```

41.2 变量

一个变量名可以由 ASCII 字符、数字和下划线组成。但是变量名不能以数字开始。以下是几个有效的变量名：

```
counter
_aap3
very_long_variable_name_with_dashes
CamelCaseName
LENGTH
```

"foo.bar" 和 "6var" 都是无效的变量名。

有些变量是全局的。要列出当前定义的所有全局变量可以用这个命令：

```
:let
```

你可以在任何地方使用全局变量。不过，太容易在两个不相关的脚本里使用相同的名字了。因此脚本里声明的变量都是局部于脚本的。例如，如果在 "script1.vim" 里有：

```
vim9script
var counter = 5
echo counter
5
```

然后试图在 "script2.vim" 里用该变量：

```
vim9script
echo counter
E121: Undefined variable: counter
```

使用脚本局部变量意味着它只能在同一脚本里改变，而不会被其它脚本影响。

如果确实要在脚本间共享变量，使用 "g:" 前缀并直接赋值，不要用 var 。使用特别名字以免出错。如在 "script1.vim" 里：

```
vim9script
g:mash_counter = 5
echo g:mash_counter
5
```

现在在 "script2.vim" 里:

```
vim9script
echo g:mash_counter
5
```

全局变量也可以从命令行访问，例如键入:

```
echo g:mash_counter
```

这不适用于脚本局部变量。

更多关于脚本局部变量可以在这里读到: script-variable 。

还有很多其它类型的变量，参阅 internal-variables 。最常用的几类有:

```
b:name          缓冲区的局部变量
w:name          窗口的局部变量
g:name          全局变量 (也用于函数中)
v:name          Vim 预定义的变量
```

删 除 变 量

变量不仅仅可以在 let 命令显示，同时也占用内存空间。为了删除全局变量，可以使用 unlet 命令。例:

```
:unlet g:counter
```

这将删除 "g:counter" 这个全局变量并释放其占用的内存。如果你并不确定这个变量是否存在，但并不希望系统在它不存在时报错，可以在命令后添加 !:

```
:unlet! g:counter
```

Vim9 脚本里不能 unlet 脚本局部变量，只有老式脚本里可以。

当一个脚本处理到结束时，它声明的局部变量不会自动被删除。脚本里定义的函数可以使用它们。例如:

```
vim9script
var counter = 0
def g:GetCount(): number
  counter += 1
  return counter
enddef
```

每次调用函数时，返回下一个计数:

```
:echo g:GetCount()
1
```

```
:echo g:GetCount()
2
```

如果担心脚本局部变量占用了太多内存而且现在不需要了，把它设为空值或 null 值。例如：

```
var lines = readfile(...)
...
lines = []
```

注意：以下例子里， vim9script 行从略，这样我们可以更加关注实际相关的命令。但你自己仍要把它放在脚本文件里。

字符串变量和常量

到目前为止我们只用到了数值作为变量的值。同样的我们可以使用字符串。这两种变量类型是 Vim 支持的基本类型。示例：

```
var name = "Peter"
echo name
Peter
```

每个变量都有类型。如此例所示，类型常常是由赋值定义的。这叫类型推导。如果你这时还不想给变量值，就需要指定类型：

```
var name: string
var age: number
if male
   name = "Peter"
   age = 42
else
   name = "Elisa"
   age = 45
endif
```

如果你搞错了，试图用错误的类型赋值，会报错：

```
age = "Peter"
E1012: Type mismatch; expected number but got string
```

类型更多的内容见 41.8 。

你需要使用字符串常量来为字符串变量赋值。字符串常量有两种。第一种是由双引号括起来的，我们已经看到过了。如果你想在这样的字符串内使用双引号，在之前加上反斜杠即可：

```
var name = "he is \"Peter\""
echo name
he is "Peter"
```

如果你不想使用反斜杠，也可以用单引号括起字符串：

```
var name = 'he is "Peter"'
echo name
```

```
he is "Peter"
```

所有的字符在单引号内都保持其本来面目。只有单引号本身例外：输入两个你会得到一个单引号。因为反斜杠在其中也被作为其本身来对待，你无法使用它来改变其后的字符的意义：

```
var name = 'P\e''ter'''
echo name
P\e'ter'
```

在双引号括起来的字符串中可以使用特殊字符。这里有一些有用的例子：

```
\t              <Tab>
\n              <NL>，换行
\r              <CR>，<Enter>
\e              <Esc>
\b              <BS>，退格
\"              "
\\              \，反斜杠
\<Esc>          <Esc>
\<C-W>          CTRL-W
```

最后两个只是用来举例子的。"\<name>" 的形式可以被用来表示特殊的键 "name"。

在 expr-quote 中列出了全部的特殊字符。

41.3 表达式

Vim 脚本支持相当标准的表达式处理。你可以在这里读到表达式的定义：
expression-syntax 。这里我们只看看常用的几个。

已经提到的那些数值，字符串和变量都属于表达式。因此任何可以使用表达式的地方，数值，字符串或变量都可以使用。其它基本的表达式有：

```
$NAME           环境变量
&name           选项值
@r              寄存器内容
```

例子：

```
echo "The value of 'tabstop' is" &ts
echo "Your home directory is" $HOME
if @a == 'text'
```

&name 这种形式也可以被用来暂时改变一个选项的值。例：

```
var save_ic = &ic
set noic
s/The Start/The Beginning/
&ic = save_ic
```

这样既确保了在匹配 "The Start" 模式时 'ignorecase' 选项是关闭的，同时也保留了用户原来的选项值。(另一个方法是在模式里加上 "\C"，见 /\C 。)

算 术

我们把这些基本的东西都混合起来用就更有趣了。先来看看算术运算：

```
a + b           加
a - b           减
a * b           乘
a / b           除
a % b           余
```

先乘除，后加减。例如：

```
echo 10 + 5 * 2
20
```

括号内的先计算。这也没什么奇怪的。例如：

```
echo (10 + 5) * 2
30
```

其 它

用 ".." 可以把两个字符串连接起来（见 expr6 ）。例如：

```
echo "Name: " .. name
Name: Peter
```

一般的，当 "echo" 命令遇到多个参数时，会在它们之间加入空格。但上例中参数是一个表达式，所以不会有空格。

如果不喜欢过多的连接，可用 $"string" 形式，花括号里接受表达式：

```
echo $"Name: {name}"
```

详细描述见 interpolated-string 。

下面的条件表达式很显然是从 C 语言里借来的：

```
a ? b : c
```

如果 "a" 为真用 "b"，否则用 "c"。例如：

```
var nr = 4
echo nr > 5 ? "nr is big" : "nr is small"
nr is small
```

在整个表达式被求值前，结构中的三部分总是先被求值的。因此你可以将其视为：

```
(a) ? (b) : (c)
```

还有准假值操作符：

```
echo name ?? "No name given"
```

见 ?? 。

41.4 条件语句

if 命令在条件满足的前提下，执行其后直到 endif 的所有语句。常用的形式为：

```
if {condition}
   {statements}
endif
```

语句 {statements} 仅当表达式 {condition} 为真或一时才被执行。不管是否执行，这些语句必须是有效的。否则 Vim 无法找到相应的 endif 。

你也可以使用 else 。常用形式为：

```
if {condition}
   {statements}
else
   {statements}
endif
```

第二组 {statements} 仅当条件不满足时被执行。

最后还有 elseif ：

```
if {condition}
   {statements}
elseif {condition}
   {statements}
endif
```

这种形式就像 else 接着 if 一样，但是少出现一个 endif 。

下面是一个有用的例子（可以用在你的 vimrc 文件里）：它检查 'term' 选项并根据不同的值做不同的操作：

```
if &term == "xterm"
  # "xterm" 对应的操作
elseif &term == "vt100"
  # vt100 终端对应的操作
else
  # 其它终端对应在操作
endif
```

这里 "#" 开始的行是注释，下面会解释。

逻 辑 操 作

实际上我们在前面的几个例子中已经是用到了。下面是几个最常用的形式：

```
a == b          等于
a != b          不等于
a >  b          大于
```

```
a >= b          大于等于
a <  b          小于
a <= b          小于等于
```

如果条件满足，结果为真，否则为假。例如:

```
if v:version >= 800
  echo "祝贺"
else
  echo "你在使用旧的版本，升级！"
endif
```

这里 "v:version" 是 Vim 定义的变量，用来存放 Vim 的版本号。800 意为 8.0 版。8.1 版的值为 801。这对编写可以在不同版本的 Vim 上运行的脚本很有用。参阅 v:version 。也可以用 has() 检查特定的特性，还可以检查特定的补丁，见 has-patch 。

对数值和字符串都可以做逻辑操作。两个字符串的算术差被用来比较它们的值。这个结果是通过字节值来计算的，对于某些语言，这样做的结果未必正确。

比较一个字符串和一个数值会报错。

对于字符串来说还有两种还有更有用的操作:

```
str =~ pat              匹配
str !~ pat              不匹配
```

左边的 "str" 被当作一个字符串。右边的 "pat" 被当作一个匹配模式，正如做查找操作一样。例如:

```
if str =~ " "
  echo "字符串包括空格"
endif
if str !~ '\.$'
  echo "字符串不以句号结尾"
endif
```

注意 在匹配模式中用单引号是很有用的。因为匹配模式中通常有很多反斜杠，而反斜杠在双引号字符串中必须双写才有效。

匹配是不定锚的，如果要匹配整个字符串，让模式以 "^" 开头，以 "$" 结尾。

在做字符串比较时不使用 'ignorecase' 选项。加上 "?" 表示忽略大小写。因此 "==?" 比较两字符串是否相等，不计大小写。 expr-== 有一个完整的列表。

循 环 详 述

while 命令已经在前面提到了。还有另外两条语句可以在 while 和 endwhile 之间使用。

```
continue                跳回 while 循环的开始；继续循环
break                   跳至 endwhile ；循环结束
```

例:

```
var counter = 1
while counter < 40
  if skip_number(counter)
    continue
  endif
  if last_number(counter)
    break
  endif
  sleep 50m
  ++counter
endwhile
```

sleep 命令使 Vim 小憩一下。"50m" 表示休息 50 毫秒。再举一个例子，`sleep 4` 休息 4 秒。

continue 和 break 也可用在 for 和 endfor 之间。更多循环可以用 for 命令实现，见下面的 41.8 。

41.5 执行一个表达式

到目前为止，脚本内的语句都是由 Vim 直接运行的。用 execute 命令可以执行一个表达式的结果。这是一个创建并执行命令的非常有效的方法。

例如要跳转到一个由变量表示的标签:

```
execute "tag " .. tag_name
```

".." 被用来连接字符串 "tag " 和变量 "tag_name" 的值。假设 "tag_name" 的值为 "get_cmd"，那么被将执行的命令将是:

```
tag get_cmd
```

execute 命令只能用来执行冒号命令。 normal 命令可以用来执行普通模式命令。然而，它的参数只能是按表面意义解释的命令字符，不能是表达式。例如:

```
normal gg=G
```

"gg" 命令将跳转到第一行，"=" 操作符和 "G" 动作排版所有行。

为了使 normal 命令也可以带表达式，可以把 execute 与其连起来使用。例:

```
execute "normal " .. count .. "j"
```

必须确保 normal 的参数是一个完整的命令。否则，Vim 碰到参数的结尾就会中止其运行。例如，如果开始了删除操作符，必须给出动作命令。这样可以:

```
normal d$
```

这样不做任何事:

```
normal d
```

如果你开始了插入模式，即使不以 Esc 结束，还是会退出插入模式。这样可以插入 "new text":

```
execute "normal inew text"
```

如果在插入文本后还要做点别的，需要退出插入模式:

```
execute "normal inew text\<Esc>b"
```

这将插入 "new text" 并把光标移到 "text" 的首字母上。注意 这里使用了特殊键 "\<Esc>"。这样就避免了在你的脚本当中键入真正的 <Esc> 字符。这时就可以看出 execute 带双引号字符串的便利之处。

如果你不想作为命令执行字符串，而想计算它作为表达式计算的结果，可以用 eval() 函数:

```
var optname = "path"
var optvalue = eval('&' .. optname)
```

"&" 被加到 "path" 前面，这样传给 eval() 的参数成为 "&path"。这时得到的返回值就是 'path' 选项的值。

41.6 使用函数

Vim 定义了大量的函数并通过这些函数提供了丰富的功能。本节将给出一些例子。你可以在下面的 function-list 找到一个完整的列表。

一个函数调用时，参数列表要用括号括起来，并用逗号分割。例如:

```
search("Date: ", "W")
```

这将以 "Date: " 和 "W" 为参数调用 search() 函数。search() 函数的第一个参数是一个查找模式，第二个参数是一个标志。标志 "W" 表示查找操作遇到文件尾时不折返。

Vim9 脚本里使用 call 命令与否可选。但老式脚本和命令行上这是必需的:

```
call search("Date: ", "W")
```

在一个表达式内也可以调用函数。例如:

```
var line = getline(".")
var repl = substitute(line, '\a', "*", "g")
setline(".", repl)
```

getline() 函数从当前缓冲区获取一行文本。其参数是行号。在本例中，"." 表示光标所在行。

substitute() 函数的功能和 :substitute 命令相似。它的第一个参数 "line" 是要执行替换操作的源字符串。第二个参数 '\a' 是一个匹配模式，第三个参数 "*" 是替换字符串。最后一个参数 "g" 是一个标志位。

setline() 函数将第一个参数表示的行的文本置为第二个参数表示的字符串。本例中光标所在的行被 substitute() 函数的结果所替换。因此这三条语句的效果等同于：

```
:substitute/\a/*/g
```

如果你在调用 substitute() 之前或之后有更多的事情要做的话，用函数的方式就会有价值了。

函 数 function-list

Vim 提供的函数很多。这里我们以它们的用途分类列出。你可以在 builtin-function-list 找到一个以字母顺序排列的列表。在函数名上使用 CTRL-] 可以跳转至该函数的详细说明。

字符串操作： string-functions

nr2char()	通过数值码值取得一个字符
list2str()	从数值列表取得字符字符串
char2nr()	取得字符的数值码值
str2list()	从字符串取得数值列表
str2nr()	把字符串转换为数值
str2float()	把字符串转换为浮点数
printf()	根据 % 项目格式化字符串
escape()	将字符串通过 '\' 转义
shellescape()	转义字符串用于外壳命令
fnameescape()	转义 Vim 命令使用的文件名
tr()	把一组字符翻译成另一组
strtrans()	将一个字符串变成可显示的格式
keytrans()	把内部键值翻译为 :map 可接受的形式
tolower()	将一个字符串转换为小写
toupper()	将一个字符串转换为大写
charclass()	字符的类
match()	字符串中的模式匹配处
matchbufline()	缓冲区里模式的所有匹配
matchend()	字符串中的模式匹配结束处
matchfuzzy()	模糊匹配字符串列表中的一个字符串
matchfuzzypos()	模糊匹配字符串列表中的一个字符串
matchstr()	在一个字符串中匹配一个模式
matchstrlist()	字符串列表中的模式的所有匹配
matchstrpos()	字符串中满足匹配的模式和位置
matchlist()	类似 matchstr()，同时返回子匹配
stridx()	子串在母串中第一次出现的地方
strridx()	子串在母串中最后一次出现的地方
strlen()	以字节计的字符串长度
strcharlen()	以字符计的字符串长度
strchars()	字符串里的字符数目
strwidth()	字符串的显示长度
strdisplaywidth()	字符串的显示长度，处理制表
setcellwidths()	设置字符单元宽度覆盖
getcellwidths()	取得字符单元宽度覆盖
getcellpixels()	取得字符单元像素尺寸
reverse()	反转字符串里的字符顺序
substitute()	用一个字符串替换一个匹配的模式
submatch()	取得 ":s" 和 substitute() 匹配中指定的某个匹配

strpart()	用字节索引取得字符串的子串
strcharpart()	用字符索引获取字符串的子串
slice()	在 Vim9 脚本中用字符索引获取字符串的切片
strgetchar()	用字符索引获取字符串里的字符
expand()	展开特殊的关键字
expandcmd()	像 :edit 那样扩展命令
iconv()	转换文本编码格式
byteidx()	字符串里字符的字节位置
byteidxcomp()	类似于 byteidx()，但计算组合字符
charidx()	字符串里字节的字符位置
utf16idx()	字符串里的字节的 UTF-16 索引
repeat()	重复字符串多次
eval()	计算字符串表达式
execute()	执行 Ex 命令并获取输出
win_execute()	类似于 execute()，但用于指定窗口
trim()	从字符串中删除字符
bindtextdomain()	设置消息查找翻译的基础路径
gettext()	查找消息翻译
ngettext()	查找单数/复数的消息翻译
str2blob()	转换字符串列表为 blob
blob2str()	转换 blob 为字符串列表

列表处理: **list-functions**

get()	得到项目，错误索引不报错
len()	列表的项目总数
empty()	检查列表是否为空
insert()	在列表某处插入项目
add()	在列表后附加项目
extend()	在列表后附加另一个列表
extendnew()	创建新列表并附加项目
remove()	删除列表里一或多个项目
copy()	建立列表的浅备份
deepcopy()	建立列表的完整备份
filter()	删除列表的选定项目
map()	改变每个列表项目
mapnew()	为改变项目创建新列表
foreach()	在列表的项目上应用函数
reduce()	缩减列表为单一值
slice()	获取列表的切片
sort()	给列表排序
reverse()	反转列表里的项目顺序
uniq()	删除重复邻接项目的备份
split()	分割字符串成为列表
join()	合并列表项目成为字符串
range()	返回数值序列的列表
string()	列表的字符串表示形式
call()	调用函数，参数以列表形式提供
index()	列表或 Blob 里某值的索引
indexof()	列表或 Blob 里表达式计算值为真的索引
max()	列表项目的最大值
min()	列表项目的最小值
count()	计算列表里某值的出现次数
repeat()	重复列表多次
flatten()	展平列表

flattennew()	展平列表的一个备份

字典处理: dict-functions

get()	得到项目，错误的键不报错
len()	字典项目的总数
has_key()	检查某键是否出现在字典里
empty()	检查字典是否为空
remove()	删除字典的项目
extend()	从一个字典增加项目到另一个字典
extendnew()	创建新字典并附加项目
filter()	删除字典的选定项目
map()	改变每个字典项目
mapnew()	为改变项目创建新字典
foreach()	在字典的项目上应用函数
keys()	得到字典的键列表
values()	得到字典的值列表
items()	得到字典的键-值组对的列表
copy()	建立字典的浅备份
deepcopy()	建立字典的完整备份
string()	字典的字符串表示形式
max()	字典项目的最大值
min()	字典项目的最小值
count()	计算字典里某值的出现次数

浮点数计算: float-functions

float2nr()	把浮点数转换为数值
abs()	绝对值（也适用于数值）
round()	四舍五入
ceil()	向上取整
floor()	向下取整
trunc()	删除小数点后的值
fmod()	除法的余数
exp()	指数
log()	自然对数（以 e 为底的对数）
log10()	以 10 为底的对数
pow()	x 的 y 次方
sqrt()	平方根
sin()	正弦
cos()	余弦
tan()	正切
asin()	反正弦
acos()	反余弦
atan()	反正切
atan2()	反正切
sinh()	双曲正弦
cosh()	双曲余弦
tanh()	双曲正切
isinf()	检查无穷
isnan()	检查非数

Blob 操作: blob-functions

blob2list()	把 blob 转换为数值列表
list2blob()	把数值列表转换为 blob
reverse()	反转 blob 里的数值顺序

其它计算: bitwise-function

and()	按位与
invert()	按位取反
or()	按位或
xor()	按位异或
sha256()	SHA-256 哈希
rand()	获取伪随机数
srand()	初始化 rand() 使用的种子

变量: var-functions

instanceof()	检查变量是否为给定类的一个实例
type()	数值形式的变量类型
typename()	文本形式的变量类型
islocked()	检查变量是否加锁
funcref()	返回指向函数的函数引用
function()	得到函数名对应的函数引用
getbufvar()	取得指定缓冲区中的变量值
setbufvar()	设定指定缓冲区中的变量值
getwinvar()	取得指定窗口的变量值
gettabvar()	取得指定标签页的变量值
gettabwinvar()	取得指定窗口和标签页的变量值
setwinvar()	设定指定窗口的变量值
settabvar()	设定指定标签页的变量值
settabwinvar()	设定指定窗口和标签页的变量值
garbagecollect()	可能情况下释放内存

光标和位置标记位置: cursor-functions mark-functions

col()	光标或位置标记所在的列
virtcol()	光标或位置标记所在的屏幕列
line()	光标或位置标记所在行
wincol()	光标所在窗口列
winline()	光标所在窗口行
cursor()	置光标于 行/列 处
screencol()	得到光标的屏幕列
screenrow()	得到光标的屏幕行
screenpos()	文本字符的屏幕行与列
virtcol2col()	屏幕上文本字符的字节索引
getcurpos()	得到光标位置
getpos()	得到光标、位置标记等的位置
setpos()	设置光标、位置标记等的位置
getmarklist()	全局/局部位置标记列表
byte2line()	取得某字节位置所在行号
line2byte()	取得某行之前的字节数
diff_filler()	得到一行之上的填充行数目
screenattr()	得到屏幕行的属性
screenchar()	得到屏幕行的字符代码
screenchars()	得到屏幕行的多个字符代码
screenstring()	得到屏幕行的字符字符串
charcol()	光标或位置标记的字符数
getcharpos()	得到光标位置标记等的字符位置
setcharpos()	设置光标位置标记等的字符位置
getcursorcharpos()	得到光标的字符位置

setcursorcharpos()　　设置光标的字符位置

操作当前缓冲区的文本：　　text-functions

getline()　　从缓冲区中取一行
getregion()　　从缓冲区里得到区域文本
getregionpos()　　从区域里得到位置列表
setline()　　替换缓冲区中的一行
append()　　附加行或行的列表到缓冲区
indent()　　某行的缩进
cindent()　　根据 C 缩进法则的某行的缩进
lispindent()　　根据 Lisp 缩进法则的某行的缩进
nextnonblank()　　查找下一个非空白行
prevnonblank()　　查找前一个非空白行
search()　　查找模式的匹配
searchpos()　　寻找模式的匹配
searchcount()　　得到在光标前/后的匹配数目
searchpair()　　查找 start/skip/end 配对的另一端
searchpairpos()　　查找 start/skip/end 配对的另一端
searchdecl()　　查找名字的声明
getcharsearch()　　返回字符搜索信息
setcharsearch()　　设置字符搜索信息

操作另一个缓冲区文本：

getbufline()　　取得指定缓冲区的行列表
getbufoneline()　　取得指定缓冲区的一行
setbufline()　　替换指定缓冲区的一行
appendbufline()　　给指定缓冲区附加行列表
deletebufline()　　从指定缓冲区中删除多行

system-functions　file-functions

系统调用及文件操作：

glob()　　展开通配符
globpath()　　在几个路径中展开通配符
glob2regpat()　　转换 glob 模式到搜索模式
findfile()　　在目录列表里查找文件
finddir()　　在目录列表里查找目录
resolve()　　找到一个快捷方式所指
fnamemodify()　　改变文件名
pathshorten()　　缩短路径里的目录名
simplify()　　简化路径，不改变其含义
executable()　　检查一个可执行程序是否存在
exepath()　　可执行程序的完整路径
filereadable()　　检查一个文件可读与否
filewritable()　　检查一个文件可写与否
getfperm()　　得到文件权限
setfperm()　　设置文件权限
getftype()　　得到文件类型
isabsolutepath()　　检查目录是否是绝对目录
isdirectory()　　检查一个目录是否存在
getfsize()　　取得文件大小
getcwd()　　取得当前工作路径
haslocaldir()　　检查当前窗口是否使用过 :lcd 或 :tcd
tempname()　　取得一个临时文件的名称
mkdir()　　建立新目录

chdir()　　改变当前目录
delete()　　删除文件
rename()　　重命名文件
system()　　得到字符串形式的外壳命令结果
systemlist()　　得到列表形式的外壳命令结果
environ()　　得到所有环境变量
getenv()　　得到一个环境变量
setenv()　　设置一个环境变量
hostname()　　系统的名称
readfile()　　读入文件到一个行列表
readblob()　　读入文件到 Blob
readdir()　　从目录得到文件名的列表
readdirex()　　从目录得到文件信息的列表
writefile()　　把一个行列表或 blob 写到文件里
filecopy()　　把 {from} 文件复制到 {to}

日期和时间:　　date-functions　time-functions
getftime()　　得到文件的最近修改时间
localtime()　　得到以秒计的当前时间
strftime()　　把时间转换为字符串
strptime()　　把日期/时间字符串转换为时间
reltime()　　得到准确的当前或者已经经过的时间
reltimestr()　　把 reltime() 的结果转换为字符串
reltimefloat()　　把 reltime() 的结果转换为浮点数

自动命令:　　autocmd-functions
autocmd_add()　　新增一组自动命令和组
autocmd_delete()　　删除一组自动命令和组
autocmd_get()　　返回自动命令的列表

缓冲区，窗口及参数列表:　　buffer-functions　window-functions　arg-functions
argc()　　参数列表项数
argidx()　　参数列表中的当前位置
arglistid()　　得到参数列表的编号
argv()　　从参数列表中取得一项
bufadd()　　给缓冲区列表增加文件
bufexists()　　检查缓冲区是否存在
buflisted()　　检查缓冲区是否存在并在列表内
bufload()　　确保缓冲区已加载
bufloaded()　　检查缓冲区是否存在并已加载
bufname()　　取得某缓冲区名
bufnr()　　取得某缓冲区号
tabpagebuflist()　　得到标签页里的缓冲区列表
tabpagenr()　　得到标签页号
tabpagewinnr()　　类似于特定标签页里的 winnr()
winnr()　　取得当前窗口的窗口号
bufwinid()　　取得某缓冲区的窗口 ID
bufwinnr()　　取得某缓冲区的窗口号
winbufnr()　　取得某窗口的缓冲区号
listener_add()　　新增回调来监听改动
listener_flush()　　调用监听器回调
listener_remove()　　删除监听器回调
win_findbuf()　　寻找包括某缓冲区的窗口

win_getid()	取得窗口的窗口 ID
win_gettype()	取得窗口的类型
win_gotoid()	转到指定 ID 的窗口
win_id2tabwin()	给出窗口 ID 获取标签页号和窗口号
win_id2win()	把窗口 ID 转换为窗口号
win_move_separator()	移动窗口的垂直分割符
win_move_statusline()	移动窗口的状态行
win_splitmove()	移动窗口成为另一个窗口的分割
getbufinfo()	获取缓冲区信息的列表
gettabinfo()	获取标签页信息的列表
getwininfo()	获取窗口信息的列表
getchangelist()	获取改变列表项目的列表
getjumplist()	获取跳转列表项目的列表
swapfilelist()	'directory' 里存在的交换文件的列表
swapinfo()	关于交换文件的信息
swapname()	取得缓冲区的交换文件路径

命令行: **command-line-functions**

getcmdcomplpat()	得到当前命令行补全模式
getcmdcompltype()	得到当前命令行补全类型
getcmdline()	得到当前命令行输入
getcmdprompt()	得到当前命令行提示符
getcmdpos()	得到命令行里的光标位置
getcmdscreenpos()	得到命令行光标的屏幕位置
setcmdline()	设置当前命令行
setcmdpos()	设置命令行里的光标位置
getcmdtype()	得到当前命令行的类型
getcmdwintype()	返回当前命令行窗口类型
getcompletion()	命令行补全匹配的列表
fullcommand()	得到完整的命令名

快速修复和位置列表: **quickfix-functions**

getqflist()	快速修复错误的列表
setqflist()	修改快速修复列表
getloclist()	位置列表项目的列表
setloclist()	修改位置列表

插入模式补全: **completion-functions**

complete()	设定要寻找的匹配
complete_add()	加入要寻找的匹配
complete_check()	检查补全是否被中止
complete_info()	取得当前补全的信息
pumvisible()	检查弹出菜单是否显示
pum_getpos()	如果可见，弹出菜单的位置及大小

折叠: **folding-functions**

foldclosed()	检查某一行是否被折叠起来
foldclosedend()	类似 foldclosed() 但同时返回最后一行
foldlevel()	检查某行的折叠级别
foldtext()	产生折叠关闭时所显示的行
foldtextresult()	得到关闭折叠显示的文本

语法和高亮: **syntax-functions** **highlighting-functions**

clearmatches()	清除 matchadd() 和 :match 诸命令定义的所有

匹配
getmatches() 得到 matchadd() 和 :match 诸命令定义的所有匹配
hlexists() 检查高亮组是否存在
hlget() 取得高亮组属性
hlset() 设置高亮组属性
hlID() 取得高亮组标示
synID() 取得某位置的语法标示
synIDattr() 取得某语法标示的特定属性
synIDtrans() 取得翻译后的语法标示
synstack() 取得指定位置的语法标示的列表
synconcealed() 取得和（语法）隐藏（conceal）相关的信息
diff_hlID() 得到比较模式某个位置的高亮标示
matchadd() 定义要高亮的模式（一个 "匹配"）
matchaddpos() 定义要高亮的位置列表
matcharg() 得到 :match 参数的相关信息
matchdelete() 删除 matchadd() 或 :match 诸命令定义的匹配
setmatches() 恢复 getmatches() 保存的匹配列表

拼写: spell-functions
spellbadword() 定位光标所在或之后的错误拼写的单词
spellsuggest() 返回建议的拼写校正列表
soundfold() 返回 "发音相似" 的单词等价形式

历史记录: history-functions
histadd() 在历史记录中加入一项
histdel() 从历史记录中删除一项
histget() 从历史记录中提取一项
histnr() 取得某历史记录的最大索引号

交互: interactive-functions
browse() 显示文件查找器
browsedir() 显示目录查找器
confirm() 让用户作出选择
getchar() 从用户那里取得一个字符输入
getcharstr() 从用户那里取得一个字符输入，以字符串形式返回
getcharmod() 取得最近键入字符的修饰符
getmousepos() 取得最近已知的鼠标位置
getmouseshape() 取得最近已知的鼠标外型
echoraw() 按原样输出字符
feedkeys() 把字符放到预输入队列中
input() 从用户那里取得一行输入
inputlist() 让用户从列表里选择一个项目
inputsecret() 从用户那里取得一行输入，不回显
inputdialog() 从用户那里取得一行输入，使用对话框
inputsave() 保存和清除预输入（typeahead）
inputrestore() 恢复预输入（译注：参阅 input()）

GUI: gui-functions
getfontname() 得到当前使用的字体名
getwinpos() Vim 窗口的位置
getwinposx() Vim 窗口的 X 位置
getwinposy() Vim 窗口的 Y 位置
balloon_show() 设置气泡的内容

```
        balloon_split()         分割消息用于气泡的显示
        balloon_gettext()       取得气泡中的文本

Vim 服务器:                                             server-functions
        serverlist()            返回服务器列表
        remote_startserver()    启动服务器
        remote_send()           向 Vim 服务器发送字符命令
        remote_expr()           在 Vim 服务器内对一个表达式求值
        server2client()         向一个服务器客户发送应答
        remote_peek()           检查一个服务器是否已经应答
        remote_read()           从一个服务器读取应答
        foreground()            将一个 Vim 窗口移至前台
        remote_foreground()     将一个 Vim 服务器窗口移至前台

窗口大小和位置:                                        window-size-functions
        winheight()             取得某窗口的高度
        winwidth()              取得某窗口的宽度
        win_screenpos()         取得某窗口的屏幕位置
        winlayout()             取得标签页中窗口的布局
        winrestcmd()            恢复窗口大小的返回命令
        winsaveview()           得到当前窗口的视图
        winrestview()           恢复保存的当前窗口的视图

映射和菜单:                                             mapping-functions
        digraph_get()           取得  digraph
        digraph_getlist()       取得所有的  digraph
        digraph_set()           注册  digraph
        digraph_setlist()       注册多个  digraph
        hasmapto()              检查映射是否存在
        mapcheck()              检查匹配的映射是否存在
        maparg()                取得映射的右部 (rhs)
        maplist()               取得所有映射的列表
        mapset()                恢复映射
        menu_info()             取得菜单项目的信息
        wildmenumode()          检查 wildmode 是否激活

测试:                                                   test-functions
        assert_equal()          断言两个表达式的值相等
        assert_equalfile()      断言两个文件的内容相同
        assert_notequal()       断言两个表达式的值不等
        assert_inrange()        断言表达式在范围内
        assert_match()          断言模式与值匹配
        assert_notmatch()       断言模式不与值匹配
        assert_false()          断言表达式为假
        assert_true()           断言表达式为真
        assert_exception()      断言命令抛出例外
        assert_beeps()          断言命令会响铃
        assert_nobeep()         断言命令不会响铃
        assert_fails()          断言命令会失败
        assert_report()         报告测试失败
        test_alloc_fail()       使内存分配失败
        test_autochdir()        启动时打开 'autochdir'
        test_override()         测试 Vim 内部的覆盖
        test_garbagecollect_now()   立即清理内存
```

test_garbagecollect_soon() 设置标志位以尽快清理内存
test_getvalue() 取得内部变量的值
test_gui_event() 生成 GUI 事件，用于测试
test_ignore_error() 忽略指定的错误信息
test_mswin_event() 生成 MS-Windows 事件
test_null_blob() 返回 null blob
test_null_channel() 返回 null 通道
test_null_dict() 返回 null 字典
test_null_function() 返回 null 函数引用
test_null_job() 返回 null 作业
test_null_list() 返回 null 列表
test_null_partial() 返回 null 偏函数
test_null_string() 返回 null 字符串
test_settime() 设置 Vim 内部使用的时间
test_setmouse() 设置鼠标位置
test_feedinput() 给输入缓冲区加入键序列
test_option_not_set() 复位指定选项已设的标志位
test_refcount() 返回表达式的引用计数
test_srand_seed() 设置 srand() 的种子值
test_unknown() 返回未知类型的值
test_void() 返回 void 类型的值

进程间通信: channel-functions
ch_canread() 检查是否有可读的内容
ch_open() 打开通道
ch_close() 关闭通道
ch_close_in() 关闭通道的 in 部分
ch_read() 从通道读取信息
ch_readblob() 从通道读取 blob
ch_readraw() 从通道读取未处理的信息
ch_sendexpr() 从通道读取 JSON 信息
ch_sendraw() 向通道发送未处理的信息
ch_evalexpr() 通过通道计算表达式
ch_evalraw() 通过通道计算未经处理的表达式
ch_status() 获取通道的状态
ch_getbufnr() 获取通道的缓冲区号
ch_getjob() 获取通道相关的作业
ch_info() 获取通道信息
ch_log() 在通道日志文件写下信息
ch_logfile() 设置通道日志文件
ch_setoptions() 设置通道的选项
json_encode() 把表达式编码为 JSON 字符串
json_decode() 把 JSON 字符串解码为 Vim 类型
js_encode() 把表达式编码为 JSON 字符串
js_decode() 把 JSON 字符串解码为 Vim 类型
base64_encode() 编码 blob 为 base64 字符串
base64_decode() 从 base64 字符串里解码 blob
err_teapot() 报错 418 或 503

作业: job-functions
job_start() 启动作业
job_stop() 停止作业
job_status() 获取作业状态
job_getchannel() 获取作业使用的通道

job_info()	获取作业信息
job_setoptions()	设置作业选项

标号：	sign-functions

sign_define()	定义或更新标号
sign_getdefined()	取得已定义的标号列表
sign_getplaced()	取得已放置的标号列表
sign_jump()	跳转到标号
sign_place()	放置标号
sign_placelist()	放置一列标号
sign_undefine()	删除标号的定义
sign_unplace()	撤销标号的放置
sign_unplacelist()	撤销一列标号的放置

终端窗口：	terminal-functions

term_start()	打开终端窗口并运行作业
term_list()	获取终端缓冲区的列表
term_sendkeys()	给终端发送键击
term_wait()	等待屏幕刷新
term_getjob()	获取终端相关联的作业
term_scrape()	获取终端屏幕的行
term_getline()	获取终端的一行文本行
term_getattr()	获取属性 {what} 的值
term_getcursor()	获取终端的光标位置
term_getscrolled()	获取终端的滚动行数
term_getaltscreen()	获取轮换屏幕标志位
term_getsize()	获取终端大小
term_getstatus()	获取终端状态
term_gettitle()	获取终端标题
term_gettty()	获取终端 tty 名
term_setansicolors()	设置 GUI 使用的 16 种 ANSI 颜色
term_getansicolors()	获取 GUI 使用的 16 种 ANSI 颜色
term_dumpdiff()	显示两份屏幕截图的差异
term_dumpload()	在窗口中载入终端屏幕截图
term_dumpwrite()	把终端屏幕的内容写入文件
term_setkill()	设置停止终端中的作业的信号
term_setrestore()	设置恢复终端的命令
term_setsize()	设置终端的大小
term_setapi()	设置终端 JSON API 函数名前缀

弹出窗口：	popup-window-functions

popup_create()	在屏幕中央创建弹出
popup_atcursor()	在光标位置正上方创建弹出，光标移开时关闭
popup_beval()	在 v:beval_ 变量指定的位置，光标移开时关闭
popup_notification()	用三秒钟显示通知
popup_dialog()	创建带填充和边框中间对齐的弹出
popup_menu()	提示从列表中选择一个项目
popup_hide()	临时隐藏弹出
popup_show()	显示之前隐藏的弹出
popup_move()	改变弹出的位置和大小
popup_setoptions()	覆盖弹出的选项
popup_settext()	替换弹出缓冲区的内容
popup_setbuf()	设置弹出缓冲区
popup_close()	关闭一个弹出

popup_clear()	关闭所有弹出
popup_filter_menu()	从一列项目中选择
popup_filter_yesno()	等待直到按了 'y' 或 'n' 为止
popup_getoptions()	取得弹出的当前选项
popup_getpos()	取得弹出的实际位置和大小
popup_findecho()	取得用于 :echowindow 的弹出的窗口 ID
popup_findinfo()	取得弹出信息窗口的窗口 ID
popup_findpreview()	取得弹出预览窗口的窗口 ID
popup_list()	取得所有的弹出窗口 ID 的列表
popup_locate()	从弹出窗口的屏幕位置取得其窗口 ID

定时器: **timer-functions**

timer_start()	建立定时器
timer_pause()	暂停或继续定时器
timer_stop()	停止定时器
timer_stopall()	停止所有定时器
timer_info()	获取定时器信息

Tags: **tag-functions**

taglist()	得到匹配标签的列表
tagfiles()	得到标签文件的列表
gettagstack()	得到窗口的标签栈
settagstack()	修改窗口的标签栈

提示缓冲区: **promptbuffer-functions**

prompt_getprompt()	得到缓冲区的实际提示文本
prompt_setcallback()	设置缓冲区的提示回调
prompt_setinterrupt()	设置缓冲区的中断回调
prompt_setprompt()	设置缓冲区的提示文本

寄存器: **register-functions**

getreg()	取得寄存器内容
getreginfo()	取得寄存器信息
getregtype()	取得寄存器类型
setreg()	设定寄存器内容及类型
reg_executing()	取得正在执行中的寄存器名
reg_recording()	取得正在记录中的突破口名

文本属性: **text-property-functions**

prop_add()	在给出位置上附属属性
prop_add_list()	在多个位置上附属属性
prop_clear()	从一行或多行中删除所有属性
prop_find()	搜索一个属性
prop_list()	返回一行中所有属性的列表
prop_remove()	从一行中删除属性
prop_type_add()	新增/定义属性类型
prop_type_change()	改变类型的属性
prop_type_delete()	删除文本属性类型
prop_type_get()	返回类型的属性
prop_type_list()	返回所有属性类型的列表

声音: **sound-functions**

sound_clear()	停止所有声音的播放
sound_playevent()	播放一个事件的声音

sound_playfile()	播放声音文件
sound_stop()	停止一个声音的播放

杂项: various-functions

mode()	取得当前编辑状态
state()	取得当前繁忙状态
visualmode()	最近一次使用过的可视模式
exists()	检查变量，函数等是否存在
exists_compiled()	类似 exists() 但在编译时检查
has()	检查 Vim 是否支持某特性
changenr()	返回最近的改变号
cscope_connection()	检查有无与 cscope 的连接
did_filetype()	检查某文件类型自动命令是否已经使用
diff()	比较两个字符串列表
eventhandler()	检查是否在一个事件处理程序内
getcellpixels()	得到单元像素尺寸的列表
getpid()	得到 Vim 的进程号
getscriptinfo()	得到执行过的 Vim 脚本列表
getstacktrace()	得到 Vim 脚本的当前栈追踪
getimstatus()	检查 IME 状态是否激活
interrupt()	中断脚本执行
windowsversion()	得到 MS-Windows 版本
terminalprops()	终端属性
libcall()	调用一个外部库函数
libcallnr()	同上，但返回一个数值
undofile()	得到撤销文件名
undotree()	返回某缓冲区的撤销树的状态
shiftwidth()	'shiftwidth' 的有效值
wordcount()	获取缓冲区的字节/单词/字符计数
id()	获取项目用于键的唯一字符串
luaeval()	计算 Lua 表达式
mzeval()	计算 MzScheme 表达式
perleval()	计算 Perl 表达式 (+perl)
py3eval()	计算 Python 表达式 (+python3)
pyeval()	计算 Python 表达式 (+python)
pyxeval()	计算 python_x 表达式
rubyeval()	计算 Ruby 表达式
debugbreak()	中断正在调试的程序

41.7 定义一个函数

Vim 允许你定义自己的函数。基本的函数声明如下:

```
def {name}({var1}, {var2}, ...): return-type
  {body}
enddef
```

注意：
函数名必须以大写字母开始。

让我们来定义一个返回两数中较小者的函数。从下面这一行开始：

```
def Min(num1: number, num2: number): number
```

这将告诉 Vim 这个函数名叫 "Min" 并且带两个数值参数："num1" 和 "num2"，并返回一个数值。

你要做的第一件事就是看看哪个数值小一些：

```
if num1 < num2
```

我们把最小的数值赋给 smaller 变量：

```
var smaller: number
if num1 < num2
  smaller = num1
else
  smaller = num2
endif
```

"smaller" 是一个局部变量。声明为数值型，这样如果搞错 Vim 就会警告。一个在函数内部使用的变量，除非被加上类似 "g:"、"w:" 或 "b:" 的前缀，都是局部变量。

备注：
为了从一个函数内部访问一个全局变量你必须在前面加上 "g:"。因此在一个函数内 "g:today" 表示全局变量 "today"，而 "today" 是另外一个仅用于该函数或脚本内的局部变量。

现在你可以使用 return 语句来把最小的数值返回给调用者了。最后，你需要结束这个函数：

```
  return smaller
enddef
```

下面是这个函数完整的定义：

```
def Min(num1: number, num2: number): number
  var smaller: number
  if num1 < num2
    smaller = num1
  else
    smaller = num2
  endif
  return smaller
enddef
```

显然这个例子太繁琐了。用两个 return 命令可以简短一些：

```
def Min(num1: number, num2: number): number
  if num1 < num2
```

```
    return num1
  endif
  return num2
enddef
```

如果你还记得条件表达式，只需要一行就够了：

```
def Min(num1: number, num2: number): number
  return num1 < num2 ? num1 : num2
enddef
```

调用用户自定义函数的方式和调用内置函数完全一致。仅仅是函数名不同而已。上面的 Min 函数可以这样来使用：

```
echo Min(5, 8)
```

只有这时函数才被 Vim 解析并执行。如果函数中有类似未定义的变量或函数之类的错误，你将得到一个错误信息。这些错误在定义函数时是不会被检测到的。要早点看到这些错误，可让 Vim 编译此脚本里的所有函数：

```
defcompile
```

编译要一点时间，但会早点报告错误。脚本编写期间，可在脚本结尾处放上 :defcompile ，一切就绪的时候再把这行注释掉。

不返回任何结果的函数只要简单省略返回类型就可以了：

```
def SayIt(text: string)
  echo text
enddef
```

如果要返回任何类型的值，可用 "any" 返回类型：

```
def GetValue(): any
```

这会关闭返回值的类型检查，除非必要不要使用。

也可以用 function 和 endfunction 定义老式函数。它们不带类型，不能编译。所以执行也会慢许多。

范围的使用

函数调用时可以带一个行表示的范围。Vim 将把光标移动到范围内的每一行，并分别对该行调用此函数。例如：

```
def Number()
  echo "line " .. line(".") .. " contains: " .. getline(".")
enddef
```

如果你用下面的方式调用该函数：

```
:10,15Number()
```

它将被执行六次，从第 10 行开始，第 15 行结束。

函 数 清 单

":function" 命令列出所有用户自定义的函数及其参数:

```
:function
def <SNR>86_Show(start: string, ...items: list<string>)
function GetVimIndent()
function SetSyn(name)
```

"<SNR>" 前缀代表函数是局部于脚本的。 Vim9 函数以 "def" 开始，包括参数和返回值类型。老式函数以 "function" 开始。

如果要查看该函数具体做什么，用该函数名作为 function 命令的参数即可:

```
:function SetSyn
1     if &syntax == ''
2       let &syntax = a:name
3     endif
   endfunction
```

要看到 "Show" 函数内容，需要提供脚本前缀，因为不同脚本可以定义多个 "Show" 函数。用 function 找到完整名字，但结果可能是个很长的列表。要只返回匹配某个模式的函数，可用 filter 前缀:

```
:filter Show function
def <SNR>86_Show(start: string, ...items: list<string>)

:function <SNR>86_Show
1    echohl Title
2    echo "start is " .. start
等等
```

调 试

调试或者遇到错误信息时，行号是很有用的。有关调试模式请参阅 debug-scripts 。

你也可以通过将 'verbose' 选项设为 12 以上来察看所有函数调用。将该参数设为 15 或以上可以查看所有被执行的行。

删 除 函 数

为了删除 SetSyn() 函数:

```
:delfunction SetSyn
```

删除只对全局函数和老式脚本里定义的函数有效， Vim9 脚本里定义的函数不行。

如果该函数不存在或不能删除，会报错。

函 数 引 用

有时使变量指向一个或另一个函数可能有用。要这么做，可用函数引用变量。常简称为"funcref"。例如：

```
def Right(): string
  return 'Right!'
enddef
def Wrong(): string
  return 'Wrong!'
enddef

var Afunc = g:result == 1 ? Right : Wrong
echo Afunc()
Wrong!
```

这里假定 "g:result" 不是一。详细描述见 Funcref 。

注意 保存函数引用的变量名必须用大写字母开头，不然和内建函数的名字会引起混淆。

进 一 步 阅 读

50.2 小节解释可变参数数目的用法。

要自定义函数，更多信息可见： user-functions 。

41.8 列表和字典

到目前为止，我们用了基本类型字符串和数值。Vim 也支持两种复合类型：列表和字典。

列表是项目的有序序列。这里的项目包括各种类型的值。所以你可以建立数值列表、列表列表甚至混合项目的列表。要建立包含三个字符串的列表：

```
var alist = ['aap', 'noot', 'mies']
```

列表项目用方括号包围，逗号分割。要建立空列表：

```
var alist = []
```

用 add() 函数可以为列表加入项目：

```
var alist = []
add(alist, 'foo')
add(alist, 'bar')
echo alist
['foo', 'bar']
```

列表的连接用 + 完成：

```
var alist = ['foo', 'bar']
alist = alist + ['and', 'more']
echo alist
['foo', 'bar', 'and', 'more']
```

或者，要用函数扩展列表，可用 extend()：

```
var alist = ['one']
extend(alist, ['two', 'three'])
echo alist
['one', 'two', 'three']
```

注意 这里如果用 add()，效果和 extend() 不一样：

```
var alist = ['one']
add(alist, ['two', 'three'])
echo alist
['one', ['two', 'three']]
```

add() 的第二个参数作为项目被加入，这样就有了嵌套的列表。

FOR 循环

使用列表的一个好处是可以在上面进行叠代：

```
var alist = ['one', 'two', 'three']
for n in alist
  echo n
endfor
one
two
three
```

这段代码循环遍历列表 "alist" 的每个项目，分别把它们的值赋给变量 "n"。for 循环通用的形式是：

```
for {varname} in {list-expression}
  {commands}
endfor
```

要循环若干次，你需要长度为给定次数的列表。range() 函数建立这样的列表：

```
for a in range(3)
  echo a
endfor
0
1
2
```

注意 range() 产生的列表的第一个项目为零，而最后一个项目比列表的长度小一。细节：内部 range() 此处并不实际产生列表，这样对循环所需的大范围的处理就更有效。对照一下在别处，range 返回实际的列表，长列表因而会需要更多时间。

你也可以指定最大值、步进，反向也可以：

```
for a in range(8, 4, -2)
  echo a
endfor
```

```
8
6
4
```

更有用的例子，遍历缓冲区中的所有行：

```
:for line in getline(1, 20)
:  if line =~ "Date: "
:    echo matchstr(line, 'Date: \zs.*')
:  endif
:endfor
```

察看行 1 到 20（包含），并回显那里找到的任何日期。

进一步的阅读可见 Lists 。

字 典

字典保存键-值组对。如果知道键，你可以快速查找值。字典用花括号形式建立：

```
var uk2nl = {one: 'een', two: 'twee', three: 'drie'}
```

现在你可以把键放在方括号里以查找单词：

```
echo uk2nl['two']
twee
```

如果键里没有特殊字母，可用句号记法：

```
echo uk2nl.two
twee
```

字典定义的通用形式是：

```
{<key> : <value>, ...}
```

空字典是不包含任何键的字典：

```
{}
```

字典的用途很多。它可用的函数也不少。例如，你可以得到它的键列表并在其上循环：

```
for key in keys(uk2nl)
  echo key
endfor
three
one
two
```

注意 这些键没有排序。你自己可以对返回列表按照特定顺序进行排序：

```
for key in sort(keys(uk2nl))
  echo key
```

```
endfor
one
three
two
```

但你永远不能得到项目定义时的顺序。为此目的，只能用列表。列表里的项目被作为有序序列保存。

进一步的阅读可见 Dictionaries 。

41.9 空白

脚本里可以使用空白行，但没有作用。

行首的空白字符（空格和制表符）总被忽略，不带 "trim" 的 :let-heredoc 是例外。

拖尾的空白常被忽略，但不绝对。 map 是这样的一个不忽略拖尾空白的命令。要小心，一旦出错很难发觉。一般建议是除非绝对必要，千万不要用拖尾空白。

为了在一个选项值内使用空格，必须像下面例子那样使用 "\" (反斜杠):

```
:set tags=my\ nice\ file
```

如果写成这样:

```
:set tags=my nice file
```

Vim 会给出错误信息，因为它被解释成:

```
:set tags=my
:set nice
:set file
```

Vim9 脚本对空白很挑剔。这是有意而为的，要确保脚本易读不易犯错。合理使用空白应该就可以了。如果有问题，会给出错误信息，告知你哪里缺空白，或者哪里有多余要删除的空白。

41.10 续行

老式 Vim 脚本里，一行的继续是在续行的行首加上反斜杠完成的:

```
let mylist = [
                \ 'one',
                \ 'two',
                \ ]
```

这需要 'cpo' 选项里排除了 "C" 标志位。通常做法是在脚本开始处放上:

```
let s:save_cpo = &cpo
set cpo&vim
```

然后在脚本结尾处恢复选项:

```
let &cpo = s:save_cpo
```

```
unlet s:save_cpo
```

更多细节可见： line-continuation 。

Vim9 脚本里还可以用反斜杠，但绝大多数情况不需要了：

```
var mylist = [
                'one',
                'two',
                ]
```

另外，也不需要修改 'cpo' 选项了。详见 vim9-line-continuation 。

41.11 注释

Vim9 脚本里 # 标记注释的开始。除了那些不接受注释的命令外（见下例），从 # 起直到行末的所有字符都将看作注释而被忽略。注释可以从一行的任意位置开始，但作为命令的一部分时除外，如字符串内的 #。

在老式脚本里 " （双引号标记）字符开始注释。需要一些小聪明来确保双引号字符串不被识别为注释（又一个推荐 Vim9 脚本的理由）。

对于某些命令来说，这里有一个小小的 "陷阱"。例如：

```
abbrev dev development  # shorthand
map <F3> o#include      # insert include
execute cmd             # do it
!ls *.c                 # list C files
```

- 缩写 'dev' 会被展开成 'development # shorthand'。
- <F3> 的键盘映射会是 'o# ....' 之后包括 '# insert include' 在内的那一整行。
- execute 命令会给出错误。
- ! 命令会将其后的所有字符传给外壳，很大可能会出错。

结论是， map ， abbreviate ， execute 和 ! 命令之后不能有注释。(另外还有几个命令也是如此)。不过，对于这些命令有一个小窍门：

```
abbrev dev development|# shorthand
map <F3> o#include|# insert include
execute '!ls *.c'       |# do it
```

'|' 字符被用来将两个命令分隔开。后一个命令仅仅是一个注释。最后一个命令里， execute 是对所有不接受注释的命令的通用解决办法，或者用 '|' 分隔下个命令。

注意 在缩写和映射后的 '|' 之前没有空格。这是因为对于这些命令，直到行尾或者 '|' 字符为止的内容都是有效的。此行为的后果之一，是你没法总看到这些命令后面包括的空白字符：

```
map <F4> o#include
```

这里确实是有意的，但其它很多情况可能只是意外。要发现这个问题，可以高亮拖尾空白：

```
match Search /\s\+$/
```

Unix 上有一个特殊的办法给一行加注释，从而使得 Vim 脚本可执行，这也适用于老式脚本：

```
#!/usr/bin/env vim -S
echo "this is a Vim script"
quit
```

41.12 文件格式

标识行尾的字符取决于系统。Vim 脚本建议总是使用 Unix 文件格式。此时行以 Newline (换行) 字符分隔。这也可用于其它系统。这样你就可以把 Vim 脚本从 MS-Windows 复制到 Unix 系统上然后还能工作。见 :source_crnl 。要确定设置正确，在写文件前做：

```
:setlocal fileformat=unix
```

如果用 "dos" 文件格式，分隔行用 CR-NL，两个字符。CR 字符会导致各种问题，最好避免使用。

编写 Vim 脚本的高级信息在 usr_50.txt 。

下一章: usr_42.txt 添加新的菜单

usr_42.txt　　适用于 Vim 9.1 版本。　　最近更新：2020年1月

VIM 用户手册 - by Bram Moolenaar
译者：lang2

添加新菜单

现在你应该已经了解到 Vim 是非常灵活的。这也包括在 GUI 中使用的菜单。你可以定义你自己的菜单项来更方便的使用一些命令。这是为那些鼠标爱好者准备的。

下一章：　usr_43.txt　使用文件类型
前一章：　usr_41.txt　编写 Vim 脚本
目录：　usr_toc.txt

42.1　简介

Vim 使用的菜单是定义在 "$VIMRUNTIME/menu.vim" 文件里的。如果你想编写自己的菜单，你可以先看看那个文件。
　使用 ":menu" 可以定义一个菜单项。这个命令的基本格式如下：

```
:menu {menu-item} {keys}
```

{menu-item} 告诉 Vim 把菜单项放在哪。"File.Save" 是一个典型的例子。它表示 "File" 菜单下的 "Save" 菜单项。那个点被用来分隔菜单的名字。例：

```
:menu File.Save  :update<CR>
```

":update" 命令在文件被修改时写入文件。
　你可以再加一层："Edit.Settings.Shiftwidth" 在 "Edit" 菜单下定义一个 "Settings" 子菜单，然后在其下定义一个菜单项 "Shiftwidth"。你还可以再加更深的层。但别用太多，那样要找到一个菜单层就需要太多的鼠标动作。
　":menu" 命令和 :map 命令很像：左边指定了该菜单项如何被触发，右边定义它将执行的字符串。{keys} 只是一串字符，Vim 会像你键入它们一样使用该字符串。因此在插入模式下，当 {keys} 不包括特殊字符时，Vim 会插入那些字符。

菜 单 快 捷 键

& 字符可以被用来指示一个菜单快捷键。例如，你可以使用 Alt-F 来选择 "File" 然后 S 来选中 "Save"。(不过 'winaltkeys' 选项可以关闭这种操作！)。因此，该菜单项的 {menu-item} 是 "&File.&Save"。快捷键使用的字符在菜单中会被加上下划线。
　你必须自己留神不要在同一菜单中使用两个一样的快捷键，那样你就不知道哪个会被选中。Vim 是不会提醒你的。

优 先 级

File.Save 真正的定义是:

```
:menu 10.340 &File.&Save<Tab>:w  :confirm w<CR>
```

数字 10.340 叫优先级数。Vim 用它来决定把该菜单放在哪里。第一个数 (10) 表示该菜单在菜单栏上的位置。优先级数较小的菜单会被放在左边，大数字对应的菜单会被放在右边。下面是标准菜单的优先级:

```
  10    20     40     50      60       70               9999

+-------------------------------------------------------------+
| File  Edit  Tools  Syntax  Buffers  Window            Help |
+-------------------------------------------------------------+
```

Help 菜单被指定了一个很大的值，这样它就会被放置在最右边。

第二个数字 (340) 决定菜单项在下拉菜单中的位置。较小的数字在上边，大的在下边。下面是 File 菜单的优先级:

```
                +-----------------+
10.310          |Open...          |
10.320          |Split-Open...    |
10.325          |New              |
10.330          |Close            |
10.335          |---------------- |
10.340          |Save             |
10.350          |Save As...       |
10.400          |---------------- |
10.410          |Split Diff with  |
10.420          |Split Patched By |
10.500          |---------------- |
10.510          |Print            |
10.600          |---------------- |
10.610          |Save-Exit        |
10.620          |Exit             |
                +-----------------+
```

注意 这些数值不是连续的。那些没有被使用的值可以被用来插入你自己定义的项。(通常情况下，最好不要改动那些标准菜单。如果需要，你可以创建新的菜单)。

当创建子菜单时，你可以加入另一级数字。这样每一个菜单项都有其优先级数。

特 殊 字 符

这里用的 {menu-item} 的例子是 "&File.&Save<Tab>:w"。这里有很重要一点: {menu-item} 必须是一个单词。如果你想加一个点，一个空格或制表符，你要么使用 <> 记法 (例如 <Space> 和 <Tab>) 或使用反斜杠。

```
:menu 10.305 &File.&Do\ It\.\.\. :exit<CR>
```

上面例子中，菜单项名 "Do It..." 包括一个空格，执行的命令是 ":exit<CR>"。

菜单名中的 <Tab> 字符可以被用来分隔菜单定义部分和用户提示部分。<Tab> 之后的部

分会被以右对齐格式显示。File.Save 菜单项的定义是 "&File.&Save<Tab>:w"。因此该菜单名是 "File.Save"；提示是 ":w"。

分 隔 符

分隔符（线）可以用来将相关的菜单放在一起。开始和最后一个字符都是 '-' 的菜单名就会认为是菜单分隔符，例如 "-sep-"。在同一个菜单中使用多个分隔符时，它们的名字必须被区别开，否则其名字不相干。

分隔符所相关的命令永远不会被执行。但你同样得定义一个。用一个冒号就行了。例如：

```
:amenu 20.510 Edit.-sep3- :
```

42.2 菜单命令

你可以为某一特定的运行模式定义专用的菜单。就好像各种 ":map" 命令的变体一样：

:menu	普通、可视和操作符等待模式
:nmenu	普通模式
:vmenu	可视模式
:omenu	操作符等待模式
:menu!	插入和命令行模式
:imenu	插入模式
:cmenu	命令行模式
:tlmenu	终端模式
:amenu	所有模式（除了终端模式外）

为了避免菜单项执行的命令被重映射，使用 ":noremenu"、":nnoremenu"、"anoremenu" 等形式。

使 用 :AMENU

":amenu" 命令有些不同。它假定你给出的 {keys} 是被用在普通模式下的。当 Vim 在可视或插入模式下运行并执行这些菜单命令时，Vim 先回到普通模式。":amenu" 会替你插入 CTRL-C 或 CTRL-O。例如，如果你使用这个命令：

```
:amenu  90.100 Mine.Find\ Word  *
```

你得到的菜单命令将会是：

```
Normal mode:            *
Visual mode:            CTRL-C *
Operator-pending mode:  CTRL-C *
Insert mode:            CTRL-O *
Command-line mode:      CTRL-C *
```

在命令行模式下 CTRL-C 会清除已键入的命令。在可视和操作符等待模式下 CTRL-C 会终止该模式。在插入模式下 CTRL-O 会执行该命令然后回到插入模式。

CTRL-O 只能用来执行一个命令。如果你要执行多于两个命令，你可以把它们定义成一个函数然后在菜单定义中调用该函数。例如：

```
:amenu  Mine.Next\ File  :call <SID>NextFile()<CR>
:function <SID>NextFile()
:  next
:  1/^Code
:endfunction
```

这个菜单项用 ":next" 命令移动到参数列表中的下一个文件，然后查找其中以 "Code" 开始的行。

函数名前的 <SID> 是一个脚本 ID。这使得该函数成为当前 Vim 脚本文件的局部函数。这是用来避免在其他脚本定义了相同名字的函数的麻烦。见 <SID> 。

默 菜 单

Vim 像你键入 {keys} 一样执行菜单命令。对于 ":" 命令这就意味着你将在命令行上看到该命令及其输出。如果命令很长的话，那个令人讨厌的 hit-Enter 提示就会出现。

为了避免这种情况，让那个菜单保持缄默。用 <silent> 参数就可以了。拿前面例子中的 NextFile() 调用来说，当你使用该菜单时，你将在命令行上看到:

```
:call <SNR>34_NextFile()
```

为了避免这些文本被显示，你可以用 "<silent>" 作为第一个参数:

```
:amenu <silent> Mine.Next\ File :call <SID>NextFile()<CR>
```

别滥用 "<silent>"。对于短的命令来说是不需要的。如果你定义一个菜单给其他人使用，让他看到被执行的命令会提示他如何只用键盘来执行这些操作而无需大费周章使用鼠标。

菜 单 列 表

当执行 menu 命令时不带 {keys} 部分，列出已经定义的菜单。你可以给出 {menu-item} 或其中的一部分，来列出指定的菜单。例如:

```
:amenu
```

这将列出所有菜单。太长了! 最好指定菜单名来减短一些:

```
:amenu Edit
```

这只列出 "Edit" 菜单下的为所有模式定义的各项。要只列出一个为插入模式定义的菜单项:

```
:imenu Edit.Undo
```

注意 要使用准确的名字。大小写是有区别的。但是 '&' 可以被省略。<Tab> 以及其后的提示也可以被省略。

删 除 菜 单

要删除一个菜单，使用 "unmenu" 命令加上和列出菜单名的命令一样的参数。这样，":menu" 变成 ":unmenu"，":nmenu" 变成 ":nunmenu"，依此类推。要删除为插入模式定

义的 "Tools.Make" 菜单项:

```
:iunmenu Tools.Make
```

你也可以用菜单名来删除包括所有子项的整个菜单。例如:

```
:aunmenu Syntax
```

这会删除 Syntax 菜单及其所有菜单项。

42.3 杂项

你可以利用 'guioptions' 选项来改变菜单的外观。缺省情况下除了 "M" 以外，所有的标志位都将被使用。你可以用下面的标志位来选择性去除菜单的某一部分。这些标志位的使用如下:

```
:set guioptions-=m
```

m	当该位被清除时整个菜单都会消失。
M	加入该位时，缺省菜单不会被加载。
g	当该位被清除时那些无效菜单会完全消失而不是仅仅被染灰(并非所有的系统都有效。)
t	当该位被清除时不使用可撕下菜单功能。

一个菜单顶部的那个用点组成的行不是分隔符。当你选中该行时，该菜单会被 "撕下"。它显示在单独的窗口里，亦称撕下菜单。当你经常使用同一菜单的时候，这是非常方便的。

要翻译菜单项，见 :menutrans 。

既然选择菜单项需要使用鼠标，最好在菜单项里使用 ":browse" 命令来选择文件；使用 ":confirm" 来获得一个对话框。这两个也可以联起来用:

```
:amenu File.Open  :browse confirm edit<CR>
```

":browse" 弹出一个文件浏览器用来选择文件。":confirm" 在当前文件改动后会自动弹出一个确认的对话框。你可以选择保存或放弃改动，或者取消命令。

对于更复杂的菜单项，可以使用 confirm() 和 inputdialog() 函数。系统缺省的菜单包含若干实例。

42.4 工具栏和弹出菜单

Vim 中有两个特殊的菜单: ToolBar (工具栏) 和 PopUp (弹出菜单)。以这两个名字开始的菜单不会出现在一般的菜单栏里。

工 具 栏

只有当 'guioptions' 选项中包括 "T" 标志位时，工具栏才会出现。
工具栏使用图标来表示一个命令，而不是文字。例如，名为 "ToolBar.New" 的 {menu-item} 会在工具栏里显示一个 "New" 图标。
Vim 有 28 个内置的图标。这里可以找到一个列表： builtin-tools 。大多数是用在缺省的工具栏里的。你可以重新定义这些工具栏项所执行的操作。(在缺省菜单被设定之后)。
你可以为工具栏项添加另外的位图，或者定义新的工具栏项。例如:

```
:tmenu ToolBar.Compile  Compile the current file
:amenu ToolBar.Compile  :!cc % -o %:r:S<CR>
```

现在你需要创建图标。对于 MS-Windows 图标必须是名为 "Compile.bmp" 的位图格式文件。对于 Unix 系统必须是名为 "Compile.xpm" 的 XPM 格式文件。大小必须是 18 X 18 个像素。对于 MS-Windows 来说也可以使用其它大小的图标，但可能看起来会比较难看。
把位图放到 'runtimepath' 其中一个目录下的 "bitmaps" 子目录内。例如: 对于 Unix 系统可以使用 "~/.vim/bitmaps/Compile.xpm" 目录。

你也可以为工具栏项定义一个工具提示。工具提示是解释工具栏项功能的简短文字。例如 "打开文件"。当鼠标移动到该工具栏项上并停留一会时，该提示会被自动显示。当图片的意图不是很明显时，这是很有用的。
例如:

```
:tmenu ToolBar.Make  Run make in the current directory
```

备注:
注意 大小写的使用。"Toolbar" 和 "toolbar" 和 "ToolBar" 是不同的!

要去除一个工具提示，可以使用 :tunmenu 命令。

'toolbar' 选项可以被用来在工具栏上显示文本而不是图标，或两者都显示。大多数人只使用位图，因为文本会占用太多空间。

弹 出 菜 单

弹出菜单会被显示在鼠标所在的位置。在 MS-Windows 系统上你点击鼠标右键就可以激活弹出菜单，然后用鼠标左键选择项目。在 Unix 系统上激活该菜单需要按住鼠标右键。
只有当 'mousemodel' 选项被设定为 "popup" 或 "popup_setpos" 时，弹出菜单才有效。两者之间的区别在于 "popup_setpos" 会将光标移动到鼠标指针所在处。当鼠标在一个选择区内时，选中的区域不受影响。当鼠标点击在选中区域外时，该选中的区域会被删除。
对于每一个运行模式分别有一个独立的弹出菜单。所以永远不会有一般菜单中的灰色菜单项。

What is the meaning of life, the universe and everything? 42
Douglas Adams, the only person who knew what this question really was about is now dead, unfortunately. So now you might wonder what the meaning of death is...

生命的意义是什么? 宇宙以及一切的一切的意义又是什么?
不幸的是，唯一了解这个问题的人，Douglas Adams 已经死了。所以你现在可能正在冥想死亡的意义是什么...
(译者注: Douglas Adams 写的科幻小说里，超级电脑对这个问题的回答是: 42。)

下一章: usr_43.txt 使用文件类型

VIM 用户手册 - by Bram Moolenaar
译者：lang2

使用文件类型

当你在编辑某一类型文件时，例如一个 C 程序或者一个外壳脚本，你通常重复使用同样的设置和键盘映射。很快你就会对每一次都要手动设置这些感到厌烦。这一章就告诉你如果自动化这些设置。

43.1 为一类文件编写的插件 filetype-plugin

如何使用文件类型插件已经在 add-filetype-plugin 这里讨论过了。不过你很可能对缺省的最基本的设定不满意。假定对于所有 C 文件你希望将 'softtabstop' 选项定为 4 并定义一个插入三行注释的键盘映射。下面的两步可以做到：

your-runtime-dir

1. 创建你自己的运行时目录。在 Unix 上通常是 "~/.vim"。在这个目录下创建 "ftplugin" 目录：

```
mkdir ~/.vim
mkdir ~/.vim/ftplugin
```

   如果你不是 Unix 用户的话，看看 'runtimepath' 选项的值就知道 Vim 在哪里找 "ftplugin" 目录了：

```
set runtimepath
```

   通常你应该使用该列表中的第一个目录名（第一个逗号之前的那个）。如果你不喜欢缺省值的话，你也可以在 vimrc 文件里把自己的目录名加到 'runtimepath' 选项的最前面。

2. 创建 "~/.vim/ftplugin/c.vim" 文件，并加入以下内容：

```
setlocal softtabstop=4
noremap <buffer> <LocalLeader>c o/**************<CR><CR>/<Esc>
let b:undo_ftplugin = "setl softtabstop< | unmap <buffer> <LocalLeader>c"
```

现在试着编辑一个 C 文件。你就会注意到 'softtabstop' 选项的值已经被设为 4 了。但是当你编辑另外的文件的时候就会被复位到 0。那是因为用了 ":setlocal" 命令。这样，对 'softtabstop' 选项的设置仅对本缓冲区有效。一旦你编辑另外一个文件，该选项的值就会被设定成那个缓冲区的缺省值，或者最近一次被 ":set" 命令设定的值。

同样地，键盘映射 "\c" 在编辑另外一个缓冲区时就不起作用了。":map <buffer>" 命令建立了一个仅对当前缓冲区有效的映射。其它映射命令 ":map!", ":vmap" 等也是如此。映射中的 <LocalLeader> 被 "maplocalleader" 变量的值所替代。

b:undo_ftplugin 的设定用于文件类型被设成其它值的时候。那种情况下你会想撤销自己的首选项。此 b:undo_ftplugin 变量作为命令执行。小心字符串里有特殊含义的字符，如反斜杠。

在下面这个目录里你可以找到一些文件类型插件的例子：

```
$VIMRUNTIME/ftplugin/
```

进一步关于为某一类文件编写插件的知识可以在这里读到： write-plugin 。

43.2 添加一个文件类型

如果你正在使用一种 Vim 不认识的文件，这一节告诉你怎么将这种文件介绍给 Vim。你需要一个自己的运行时目录。参阅上面的 your-runtime-dir 。

创建一个文件 "filetype.vim" 并加入一个为你的文件类型编写的自动命令。(关于自动命令的阐述在 40.3 。) 例：

```
augroup filetypedetect
au BufNewFile,BufRead *.xyz     setf xyz
augroup END
```

这样所有以 ".xyz" 结尾的文件将被认为 "xyz" 类型的文件。":augroup" 命令将该自动命令加入到 "filetypedetect" 组。这样做的作用是：当用户用 ":filetype off" 命令的命令时，所有文件类型检测的自动命令都被忽略掉。"setf" 命令将 'filetype' 选项设为该命令的参数，除非该选项已经被设置过。该命令保证 'filetype' 不会被重复设定。

你可以使用各种各样的模式来匹配你的文件名。也可以包括目录名。见 autocmd-patterns 。例如，"/usr/share/scripts/" 目录下的文件都是 "ruby" 文件，但没有文件扩展名。加入如下一行就可以了：

```
augroup filetypedetect
au BufNewFile,BufRead *.xyz                     setf xyz
au BufNewFile,BufRead /usr/share/scripts/*      setf ruby
augroup END
```

然而，如果你编辑一个叫做 /usr/share/scripts/README.txt 的文件，那可不是 ruby 文件。使用以 "*" 结尾的模式的不妥之处就在于它会匹配过多的文件。为了避免这种情况，把那个 filetype.vim 文件放到位于 'runtimepath' 最后的那个目录。以 Unix 为例，你可以用 "~/.vim/after/filetype.vim"。

现在你可以把文本文件的检测加入 ~/.vim/filetype.vim:

```
augroup filetypedetect
au BufNewFile,BufRead *.txt                     setf text
augroup END
```

'runtimepath' 首先找到该文件。最后才是 ~/.vim/after/filetype.vim:

```
augroup filetypedetect
au BufNewFile,BufRead /usr/share/scripts/*      setf ruby
augroup END
```

Vim 会在每一个 'runtimepath' 列出的目录中查找 "filetype.vim" 文件。先是 ~/.vim/filetype.vim。匹配 *.txt 文件的自动命令是在那里定义的。接着 Vim 找到 $VIMRUNTIME 中的 filetype.vim，因为该目录在 'runtimepath' 的中部。最后才找到 ~/.vim/after/filetype.vim，然后其中检测 ruby 文件的自动命令才被添加到系统中。

现在你在编辑 /usr/share/scripts/README.txt 时，自动命令是以其被定义的次序一一检测的。因为匹配 *.txt 模式，这样就会执行 "setf text" 命令，将文件类型设定为 "text"。之后 ruby 文件的模式也匹配了，"setf ruby" 被执行。但是因为 'filetype' 已经被设为 "text" 了，后者就什么作用也不起了。

当你编辑文件 /usr/share/scripts/foobar 同样的自动命令被检测。但只有 ruby 文件的那个匹配，因此 "setf ruby" 命令将 'filetype' 设为 ruby。

依 内 容 而 定

如果你的文件无法以其文件名决定类型，你可以通过其内容来确定。例如，很多脚本文件都是这样开始的：

```
#!/bin/xyz
```

为了认出这个脚本，在你的运行时目录（和你的 filetype.vim 在同一个目录）下创建一个 "scripts.vim" 文件。类似这样：

```
if did_filetype()
  finish
endif
if getline(1) =~ '^#!.*[/\\]xyz\>'
  setf xyz
endif
```

先用 did_filetype() 可以避免你无谓的检查已经被检测出文件类型的文件。这样就不必浪费时间了："setf" 一点作用也没有。

scripts.vim 文件被缺省的 filetype.vim 文件中的一个自动命令调用。因此，检测的次序是：

1. 'runtimepath' 内 $VIMRUNTIME 之前目录中的 filetype.vim 文件
2. $VIMRUNTIME/filetype.vim 的前部
3. 'runtimepath' 内所有的 script.vim 文件
4. $VIMRUNTIME/filetype.vim 余下的部分
5. 'runtimepath' 内 $VIMRUNTIME 之后目录中的 filetype.vim 文件

如果这样对你还是不够的话，加入一个匹配所有文件的自动命令然后调用一个脚本或者执行一个函数来检查文件的内容。

下一章： usr_44.txt 自定义语法高亮

usr_44.txt　　适用于 Vim 9.1 版本。　　最近更新：2017年8月

VIM 用户手册 - by Bram Moolenaar
译者：wandys、tocer

自定义语法高亮

Vim 对上百种文件都有自动语法高亮。如果你所编辑的文件类型没有被包括，通过阅读本章，你可以掌握对这种文件类型自定义语法高亮的方法。同时请参考　:syn-define 。

下一章：　usr_45.txt　选择你的语言
前一章：　usr_43.txt　使用文件类型
目录：　usr_toc.txt

44.1　基本语法命令

利用一个已经存在的语法文件可以节省很多时间。首先看 $VIMRUNTIME/syntax 目录下是否存在一种类似语言的语法文件。这些文件同时可以让你了解语法文件的一般布局。为了进一步理解，你需要阅读以下的部分。

让我们先从简单的命令开始。在定义新的语法之前，我们需要清除旧的定义：

```
:syntax clear
```

这在最终的语法文件中不是必须的，但在我们试验时却很有用。

本章有很多简化。如果你要编写一个供别人使用的语法文件，请从头到尾阅读本章以通晓有关细节。

列 出 已 定 义 的 项 目

要查看现在已定义的项目，用这个命令：

```
:syntax
```

你可以用此命令查看哪些命令确实已被定义。这在调试新的语法文件时很有用。它还会显

示每个项目所用的颜色，可以更好的区分彼此。
列出某个特定语法组的项目，用:

```
:syntax list {group-name}
```

这还可以用来列出簇（在 44.8 有解释)。只需要在名字中包含 '@'。

匹 配 大 小 写

一些语言是大小写不敏感的，比如 Pascal。另外一些，比如 C，则是大小写敏感的。你需要用以下命令指出你的类型:

```
:syntax case match
:syntax case ignore
```

参数 "match" 意味着 Vim 区分语法元素的大小写，这样 "int" 和 "Int" 与 "INT" 不同。如果指定 "ignore" 参数，"Procedure"、"PROCEDURE" 和 "procedure" 均视为相同。
":syntax case" 命令可以出现在语法文件的任何地方，并对其后的语法定义产生效用。在大多数情况下，你只需要一个 ":syntax case" 命令；但如果你使用一种不常见的语言，这种语言既包括大小写敏感的元素又包括大小写不敏感的元素，你需要在文件中使用多个 ":syntax case" 命令。

44.2 关键字

最基本的语法元素是关键字。要定义一个关键字，用以下命令:

```
:syntax keyword {group} {keyword} ...
```

{group} 是语法组的名字。用 ":highlight" 命令，你可以对 {group} 设定颜色。{keyword} 参数是关键字。这里有一些例子:

```
:syntax keyword xType int long char
:syntax keyword xStatement if then else endif
```

这个例子中用到了两个名为 "xType" 和 "xStatement" 的语法组。按惯例，每个组名都用所定义语言的文件类型作为前缀。本例中是为 x 语言（样例语言（eXample)，无特殊含义）定义语法。在为 "csh" 脚本建立的语法文件中，将用到类似 "cshType" 的名字。这样，前缀和 'filetype' 的值相同。
这些命令使 "int"，"long"，"char" 以一种方式被高亮显示，而 "if"、"then"、"else" 和 "endif" 则以另一种方式被高亮显示。现在你需要将 x 组名和标准的 Vim 名字联系起来:

```
:highlight link xType Type
:highlight link xStatement Statement
```

这告诉 Vim 以 "Type" 的方式高亮显示 "xType"，以 "Statement" 的方式高亮显示 "xStatement"。关于标准名，参考 group-name 。

非 常 见 关 键 字

在关键字里用到的字符必须在 'iskeyword' 选项内。如果你使用其它的字符，关键字将

不被匹配。Vim 对此并不给出警告信息。
x 语言在关键字中用到了 '-' 字符。使用的方法如下:

```
:setlocal iskeyword+=-
:syntax keyword xStatement when-not
```

"setlocal" 命令用来指定只对当前缓冲区改变 'iskeyword'。但仍会对 "w" 和 "*" 等命令造成影响。如果这不是所期望的，不要定义关键字而使用匹配 (在下一节内解释)。

x 语言允许缩写。比如，"next" 可以缩写成 "n"、"ne" 或者 "nex"。你可以用这个命令来定义:

```
:syntax keyword xStatement n[ext]
```

这并不会匹配 "nextone"，关键字只匹配完整的单词。

44.3 匹配

试想一下定义更复杂的东西。你想要匹配普通的标识符。对此，你定义一个匹配语法项目。它会匹配任意一个只由小写字母组成的单词:

```
:syntax match xIdentifier /\<\l\+\>/
```

备注:
关键字超越其它的语法项目。这样上面的 ":syntax keyword" 命令定义的 "if"，"then" 等关键字将被视为关键字，即使它们匹配 xIdentifier 的模式。

最后一部分是一个模式 (类似查找中的模式)。// 用来包含模式 (像 ":substitute" 命令中的一样)。你也可以用其它的字符，比如加号或者引号。

现在定义一个注释的匹配。在 x 语言中注释是以 # 开头的行:

```
:syntax match xComment /#.*/
```

你可以用任意的查找模式。因此通过一个匹配项目，你可以高亮显示非常复杂的项目。关于查找模式，参考 pattern 。

44.4 区域

在 x 样例语言中，字符串用双引号 (") 来包含。要高亮一个字符串，你需要定义一个区域。你需要一个区域的开头 (双引号) 和一个区域的结尾 (双引号)。定义如下所示:

```
:syntax region xString start=/"/ end=/"/
```

"start" 和 "end" 指示定义用于寻找区域开头和结尾的模式。但如果一个有这样的字符串该怎么办呢?

```
"A string with a double quote (\") in it"
```

这会带来一个问题: 字符串中间的双引号会结束区域。你需要告诉 Vim 忽略任何转义之后的双引号:

```
:syntax region xString start=/"/ skip=/\\"/ end=/"/
```

用两个反斜杠匹配单个反斜杠，因为反斜杠在字符串中是一个特殊字符。

什么时候用区域而不用匹配？两者主要的区别是匹配是单个的模式，它必须作为整体来匹配；而区域是只要区域的 "开始" 匹配便开始匹配，"结尾" 匹配是否找到并不重要。这样当一个项目倚赖于 "结尾" 匹配时，你不能用区域。区域定义起来更简单些，当使用嵌套项目时也更加方便，这将在下一节中解释。

44.5 嵌套项目

看一下这个注释：

```
%Get input  TODO: Skip white space
```

你想将 "TODO" 高亮显示成黄色字符，即使它在一个蓝色高亮显示的注释中。要使 Vim 了解到这些，你需要定义以下的语法组：

```
:syntax keyword xTodo TODO contained
:syntax match xComment /%.*/ contains=xTodo
```

在第一行中，"contained" 参数告诉 Vim 这个关键字只能存在于另一个语法项目中。下一行指定 "contains=xTodo"，这说明 xTodo 语法元素存在其中。结果就是注释行作为整体匹配 "xComment" 并被显示成蓝色。而其中的 "TODO" 单词匹配 "xTodo" 将被显示成黄色。

递 归 嵌 套

x 语言用大括号定义代码块。而且一个代码块中可以包含其它的代码块。这可以用以下方式定义：

```
:syntax region xBlock start=/{/ end=/}/ contains=xBlock
```

假设你有这样的代码：

```
while i < b {
        if a {
                b = c;
        }
}
```

首先一个 xBlock 开始于第一行的 {。第二行中有另一个 {。因为我们已经在一个 xBlock 之内，因此一个嵌套的 xBlock 在此处开始。这样 "b = c" 那一行在第二级的 xBlock 区域之内。下一行有一个 }，这将和区域的结尾相匹配。它会结束嵌套的 xBlock。因为这个 } 存在于嵌套的区域，它在第一个 xBlock 区域的视野里被隐藏。因此最后一行的 } 结束第一个 xBlock 区域。

保 留 结 尾

考虑下面的两个语法项目：

```
:syntax region xComment start=/%/ end=/$/ contained
:syntax region xPreProc start=/#/ end=/$/ contains=xComment
```

你定义注释是以 % 开始的任意行，而预处理命令是以 # 开始的任意行。因为预处理行可以有注释，预处理的定义包含一个 "contains=xComment" 参数。现在看一下下面的代码会发生什么情况：

```
#define X = Y  % Comment text
int foo = 1;
```

你看到的是第二行也被当作预处理高亮显示。预处理命令应该在行结尾处结束。这也是你要使用 "end=/$/" 的原因。但到底是哪里错了呢？

问题在于被包含的注释。注释以 % 开始，在行尾结束。注释结束后，预处理语法继续，这是在行尾已被处理之后进行的，因此下一行也被认为是预处理命令。

为避免这个问题，避免一个包含在内的语法项目吃掉一个必需的行尾，使用 "keepend" 参数。这样就会解决两个行结尾的匹配问题：

```
:syntax region xComment start=/%/ end=/$/ contained
:syntax region xPreProc start=/#/ end=/$/ contains=xComment keepend
```

包含多个项目

你可以用 contains 参数来指定任何项目都能被包含。比如：

```
:syntax region xList start=/\[/ end=/\]/ contains=ALL
```

所有的项目都包含在此项中。它也包含它自身，但不能在同一位置（这是为了避免无限循环）。

你可以指定不被包括的组，这样就包含除了指定组之外的所有组：

```
:syntax region xList start=/\[/ end=/\]/ contains=ALLBUT,xString
```

使用 "TOP" 项目，你可以包含所有那些没有 "contained" 参数的项目。"CONTAINED" 用来只包含那些使用了 "contained" 参数的项目。更多信息，参考 :syn-contains 。

44.6 跟随组

x 语言有这种形式的语句：

```
if (condition) then
```

你想对这三种项目使用不同的高亮。尽管 "(condition)" 和 "then" 可能在其它地方有其它的高亮显示方式。这时你需要这样做：

```
:syntax match xIf /if/ nextgroup=xIfCondition skipwhite
:syntax match xIfCondition /([^)]*)/ contained nextgroup=xThen skipwhite
:syntax match xThen /then/ contained
```

"nextgroup" 参数指定跟随的项目。这并不是必须的。如果指定的项目都没找到，什么事情都不会发生。比如，在下面的代码中：

```
if not (condition) then
```

"if" 匹配 xIf。"not" 并不匹配指定的跟随组 xIfCondition，这样只有 "if" 被高亮显示。

"skipwhite" 参数告诉 Vim 空白字符（空格和制表符）可以出现在项目之间。相似地，"skipnl" 参数允许项目之间存在换行符；"skipempty" 允许存在空行。注意 "skipnl" 并不忽略一个空行，换行符之后必须有能够匹配的内容。

44.7 其它参数

MATCHGROUP

当你定义一个区域时，整个区域将根据指定的组名被高亮显示。比如，要高亮显示 xInside 组被小括号 () 包含的部分，用下面的命令：

```
:syntax region xInside start=/(/ end=/)/
```

假设，你要以不同方式显示括号。你可以用很多复杂的区域语句来完成，或者你可以使用 "matchgroup" 参数。它将告诉 Vim 用另外一个组的方式来显示区域的开头和结尾（在本例中是 xParen 组）：

```
:syntax region xInside matchgroup=xParen start=/(/ end=/)/
```

"matchgroup" 参数适用于它之后的区域开头或结尾。在前一例中，开头和结尾都以 xParen 方式高亮显示。如果要用 xParenEnd 显示结尾：

```
:syntax region xInside matchgroup=xParen start=/(/
        \ matchgroup=xParenEnd end=/)/
```

用 "matchgroup" 的一个副作用是被包含的项目不会在区域的开头或结尾匹配。"transparent" 的例子中用到此作用。

TRANSPARENT

在 C 语言文件中你想要以不同方式高亮显示 "while" 和 "for" 后面的小括号。这两者中均会出现嵌套的小括号，它们会以相同的方式显示。你必须保证括号的高亮显示在匹配的 ')' 处结束。下面是一种可用的方法：

```
:syntax region cWhile matchgroup=cWhile start=/while\s*(/ end=/)/
        \ contains=cCondNest
:syntax region cFor matchgroup=cFor start=/for\s*(/ end=/)/
        \ contains=cCondNest
:syntax region cCondNest start=/(/ end=/)/ contained transparent
```

你现在可以用不同方式高亮显示 cWhile 和 cFor。cCondNest 项目可以出现在它们中的任何一个，并且使用所处那个项目的高亮显示。这是 "transparent" 参数产生的效果。

注意 "matchgroup" 参数和项目本身是同一组。为什么要定义它呢？使用 "matchgroup" 的副作用是被包含的项目并不会在开始项目之中寻找匹配。这样就避免了 cCondNest 组匹配 "for" 或 "while" 后面立即跟随的 (。否则，它会扫描整个文本直到匹配的)，而区域从其后继续。现在 cConNest 只在开始模式匹配后才开始匹配，就是在

第一个 (之后。

位 移

假设你要定义一个在 "if" 之后并在 (和) 之间的区域。但是你不想包括 "if" 或者 (和)。你可以为模式定义一个位移。比如:

```
:syntax region xCond start=/if\s*(/ms=e+1 end=/)/me=s-1
```

开始模式的位移是 "ms=e+1"。"ms" 代表匹配开始 (Match Start)。它定义模式开始的偏移量。"e+1" 的意思是匹配在模式结束处 (End) 后再忽略一个字符后开始匹配。
结尾模式的位移是 "me=s-1"。"me" 表示匹配结尾 (Match End)。"s-1" 的意思是在模式开始 (Start) 的前面一个字母处。下面这个代码的结果是:

```
if (foo == bar)
```

只有 "foo == bar" 部分以 xCond 方式高亮显示。

关于偏移量的更多信息，参考: :syn-pattern-offset 。

ONELINE

"oneline" 参数说明区域不超过一行的范围。比如:

```
:syntax region xIfThen start=/if/ end=/then/ oneline
```

这定义了一个开始于 "if" 并在 "then" 结束的区域。但如果在 "if" 后没有 "then"，这个区域就不会被匹配。

> 备注:
> 当使用 "oneline" 时，如果结尾模式没有在同一行被匹配，那么整个区域就不会匹配。如果不指定 "oneline"，Vim 不会 检查结尾模式是否匹配；即使结尾模式在文件后面的部分没有匹配，区域也将开始匹配。

续 行 及 避 免

现在事情变得有点复杂。让我们来定义一个预处理行。它以 # 开始到行尾结束。一个以 \ 结束的行使得下一行成为它的续行。这样你可以允许语法项目包括一个续行模式:

```
:syntax region xPreProc start=/^#/ end=/$/ contains=xLineContinue
:syntax match xLineContinue "\\$" contained
```

在本例中，虽然 xPreProc 一般只匹配单个行，它包含的组 (xLineContinue) 允许它使用多行。比如，它会匹配下面的两行:

```
#define SPAM  spam spam spam \
                bacon and spam
```

在本例中，这就是你所需要的。如果它并不是你想要的，你可以通过给被包含的模式添加 "excludenl" 参数来使区域在一个单行上。比如，你想高亮显示 xPreProc 中只在行结尾处的 "end"。为了避免 xPreProc 像 xLineContinue 那样扩展到下一行，使用

"excludenl":

```
:syntax region xPreProc start=/^#/ end=/$/
        \ contains=xLineContinue,xPreProcEnd
:syntax match xPreProcEnd excludenl /end$/ contained
:syntax match xLineContinue "\\$" contained
```

"excludenl" 必须写在模式之前。因为 xLineContinue 并没有 "excludenl"，它的匹配会像前面那样扩展 xPreProc 到下一行。

44.8 簇

开始编写语法文件时，你会注意到你要产生很多的语法组。Vim 能让你定义一个语法的集合 -- 簇 (cluster)。

假设你有一个语言，它含有 "for" 循环，"if" 语句，"while" 循环和函数。它们都含有相同的语法元素：数字，标识符。你可以这样定义它们：

```
:syntax match xFor /^for.*/ contains=xNumber,xIdent
:syntax match xIf /^if.*/ contains=xNumber,xIdent
:syntax match xWhile /^while.*/ contains=xNumber,xIdent
```

你需要重复使用 "contains" 参数。如果你想添加另一个包含项目，你需要把它加三次。语法簇简化了这些定义，你可以使用一个簇来代表这些语法组。

要为这三个组包含的两个项目定义一个簇，用下面的命令：

```
:syntax cluster xState contains=xNumber,xIdent
```

簇可以像语法组那样用在其它的项目内。它们的名字以 '@' 开头。因此你可以这样定义这三个组：

```
:syntax match xFor /^for.*/ contains=@xState
:syntax match xIf /^if.*/ contains=@xState
:syntax match xWhile /^while.*/ contains=@xState
```

你可以用 "add" 参数给簇添加新组：

```
:syntax cluster xState add=xString
```

你也可以用 "remove" 参数从簇里删除某个组：

```
:syntax cluster xState remove=xNumber
```

44.9 包含其它语法文件

C++ 语言的语法是 C 语言的超集。因为你不想编写两个独立的语法文件，你可以在 C++ 语法文件中首先读入 C 语言的语法文件：

```
:runtime! syntax/c.vim
```

":runtime!" 命令会在 'runtimepath' 里搜索所有的 "syntax/c.vim" 文件。这使得 C++ 语法中的 C 部分定义的方式就像 C 文件里使用的那样。如果你替换了 c.vim 语法

文件，或是在另一个文件中增加了项目，这些都会被载入。
载入 C 语法项目之后，可以定义特定的 C++ 项目。比如，添加一些没在 C 中用到的关键字：

```
:syntax keyword cppStatement    new delete this friend using
```

这会像任何其它的语法文件一样起作用。

现在考虑一下 Perl 语言。Perl 脚本包括两个不同的部分：POD 格式的文档和用 Perl 本身编写的程序。POD 部分以 "=head" 开头并以 "=cut" 结尾。
你想要在一个文件中定义 POD 语法，并在 Perl 语法文件中使用它。
":syntax include" 命令读入一个语法文件并将定义的元素储存到一个语法簇中。对 Perl 来说，格式如下所示：

```
:syntax include @Pod <sfile>:p:h/pod.vim
:syntax region perlPOD start=/^=head/ end=/^=cut/ contains=@Pod
```

perlPOD 开始于 Perl 文件中的 "=head"。这个区域包含 @Pod 簇。所有在 pod.vim 语法文件定义的顶层项目均在这里匹配。当找到 "=cut"，区域结束，我们回到在 Perl 文件中定义的项目。
":syntax include" 命令很聪明，它会忽略包含文件中的 ":syntax clear"。并且类似于 "contains=All" 的参数只会包括存在于被包含文件的项目，而不会包括包含文件内的项目。
"<sfine>:p:h/" 部分使用当前文件的名字 (<sfile>)，把它展开成全路径 (:p)，然后取头部分 (:h)。结果就是文件的目录名。这将包含相同目录下的 pod.vim 文件。

44.10 同步

编译器处理起这些来很简单。它们在文件头开始解析以至文件尾。Vim 并没有这么简单。它必须从编辑进行的中间开始。那么怎么确定它的位置？
秘密在于 ":syntax sync" 命令。它告诉 Vim 怎样找出当前位置。比如，下面的命令告诉 Vim 向后扫描 C 风格注释的开头和结尾，并对注释高亮显示：

```
:syntax sync ccomment
```

你可以使用其它的参数。"minlines" 参数告诉 Vim 向后扫描的最小行数；"maxline" 则指定最大行数。
比如，下面的命令告诉 Vim 从屏幕的顶端开始反向扫描至少 10 行：

```
:syntax sync ccomment minlines=10 maxlines=500
```

如果它不能在这个范围内确定位置，它会继续反向查看直至它知道要怎样做。但它最多就查看 500 行。(大的 "maxline" 值会降低速度；而小的值会使同步失败。)
为使同步更快，告诉 Vim 哪些语法项目可以被忽略。对那些只有当确实要显示时才用到的匹配和区域，可以指定 "display" 参数。
缺省情况下，找到的注释会以 Comment 语法组指定的方式显示颜色。如果你想用其它的颜色显示，你可以指定一个不同的语法组：

```
:syntax sync ccomment xAltComment
```

如果你的语言没有 C 风格的注释，你可以用其它的同步方式。最简单的方法是告诉 Vim 跳过若干行并从那里开始定位。下面的命令指定 Vim 反向跳过 150 行，并从那里开始分析：

```
:syntax sync minlines=150
```

一个大的 "minlines" 会使 Vim 变得很慢，尤其是向后卷屏的时候。
最后，你可以用这个命令来指定要查找的语法组：

```
:syntax sync match {sync-group-name}
        \ grouphere {group-name} {pattern}
```

这会告诉 Vim 当它看到 {pattern}，名为 {group-name} 的语法组会在其后开始。{sync-group-name} 用于给这个同步规格说明一个名字。比如，外壳脚本的 if 语句以 "if" 开头，以 "fi" 结尾。

```
if [ --f file.txt ] ; then
        echo "File exists"
fi
```

你可以用下面的命令为这个语法定义一个 "grouphere" 指示：

```
:syntax sync match shIfSync grouphere shIf "\<if\>"
```

"groupthere" 参数告诉 Vim 某个模式结束一个组。比如，if/fi 组的结束如下所示：

```
:syntax sync match shIfSync groupthere NONE "\<fi\>"
```

在这个例子中，NONE 告诉 Vim 你不在任何一个特殊的区域内。特别地，你不在一个 if 代码块内。

你还可以定义没有 "grouphere" 或 "groupthere" 参数的匹配和区域。这些组在同步时会被忽略。比如，下面的命令忽略 {} 内的所有部分，即使它们通常能在其它的同步方法中匹配：

```
:syntax sync match xSpecial /{.*}/
```

有关同步的更多信息，请见参考手册： :syn-sync 。

44.11 安装语法文件

当你新的语法文件要投入使用时，把它放到在 'runtimepath' 下的一个 "syntax" 目录。对 Unix 来讲，它可以是 "~/.vim/syntax"。
语法文件的名字必须要和文件类型相同并以 ".vim" 做后缀。这样，对 x 语言来讲，这个文件的完整路径是：

```
~/.vim/syntax/x.vim
```

你要确保该文件类型可被识别。参考 43.2 。

如果你的语言文件工作的很好，你也许想要其他的 Vim 用户来使用。首先阅读下一节以确保你的文件能很好的适用于别人。然后将它 email 给 Vim 维护者：<maintainer@vim.org>。并要解释文件类型是怎样识别的。如果幸运的话，你的文件会包括在 Vim 的下一版本中！

向 已 有 文 件 添 加 项 目

我们前边是假定你要写一个全新的语法文件。如果一个已有的语法文件可以工作但是缺少某些项目，你可以将这些项目添加到另一个独立的文件。这样不用改变发布的语言文件，因为安装 Vim 的新版本时会丢失这些改变。

在你的文件中写入语法命令（也许要用到已有语法的组名）。比如，给 C 语法文件添加新的变量类型：

```
:syntax keyword cType off_t uint
```

文件应使用和原始语法文件一样的名字。在本例中，是 "c.vim"。把它放到 'runtimepath' 里的靠后的目录里。对 Unix 而言，该目录可以是：

```
~/.vim/after/syntax/c.vim
```

44.12 可移植语法文件格式

如果所有的 Vim 用户都能交换语法文件岂不更好？为此，语法文件必须符合一些规则。

首先要有一个说明语法文件目的，维护人，以及更新时间的文件头。但不要包含太多的历史更改信息。比如：

```
" Vim syntax file
" Language:     C
" Maintainer:   Bram Moolenaar <Bram@vim.org>
" Last Change:  2001 Jun 18
" Remark:       Included by the C++ syntax.
```

用和其它语法文件相同的格式。使用一个已有的语法文件作为样例会节约很多时间。

为你的语法文件选择一个好的，描述性强的名字。使用小写字符和数字。不要使用太长的名字，因为它将在很多地方用到：语法文件名 "name.vim"，'filetype'，b:current_syntax 以及组名的开头（例如：nameType、nameStatement、nameStrings 等）。

以一个 "b:current_syntax" 检查开头。如果它被定义，表明其它语法文件（在 'runtimepath' 前部的）已被载入：

```
if exists("b:current_syntax")
  finish
endif
```

在最后将 "b:current_syntax" 设定为语法的名字。不要忘了被包含的文件也会这样做，当你包含两个文件时，也许要复位 "b:current_syntax"。

不要包括任何的用户首选项。不要设定 'tabstop'，'expandtab' 等。这些属于文件类型脚本。

不要包括映射或者缩写。如果确实需要识别关键字，只设定 'iskeyword'。

为使用户能够选择自己喜欢的颜色，请为每一种类型的高亮项目制作不同的高亮组。然后把它们链接到标准高亮组。这使每个色彩方案能正常工作。如果你选择特定的颜色，那样会使一些色彩方案看起来很难看。不要忘了一些人使用不同的背景，或者只能使用八种颜

色。

链接可用 "hi def link"，这样用户就可以在语法文件载入前选择不同的高亮。例如：

```
hi def link nameString        String
hi def link nameNumber        Number
hi def link nameCommand       Statement
... 等等 ...
```

给那些在同步时用不到的项目添加 "display" 参数可以提高向后滚屏和 CTRL-L 的速度。

下一章： usr_45.txt 选择你的语言

usr_45.txt　　适用于 Vim 9.1 版本。　　最近更新：2025年3月

VIM 用户手册 - by Bram Moolenaar
译者：lang2

选择你的语言（locale）

Vim 中各种的消息可以用多种不同的语言给出。本章就介绍如何改变使用的语言。同时还讲解如何编辑各种语言的文件。

下一章：　usr_50.txt　　高级 Vim 脚本编写
前一章：　usr_44.txt　　自定义语法高亮
目录：　usr_toc.txt

45.1　消息所用语言

当你启动 Vim 时，它会检查环境变量从而找出你所使用的语言。大多数情况下这就够了。你会看到你所用语言的消息（如果有的话）。要查看当前语言，使用命令：

```
:language
```

如果你得到 "C" 的输出，表示 Vim 在使用缺省的语言 -- 英语。

备注：
只有当多种语言支持被编译进 Vim 时，你才可能用它来编辑各种不同的语言。要找出 Vim 是否支持多语种，看一下 ":version" 命令的输出中有没有 "+gettext" 和 "+multi_lang"。如果有的话，表示应该没问题了。如果你看到 "-gettext" 或者 "-multi_lang" 的话你得重找一个 Vim。

如果你想要 Vim 的输出消息也用另外一种语言呢？有几种可能的办法。你应该使用哪一种取决于你所使用的系统。
　第一种办法是在启动 Vim 前设置环境变量。以 Unix 为例：

```
env LANG=de_DE.ISO_8859-1  vim
```

只有你的系统支持该语言的时候这才有效。这个办法的好处是所有的 GUI 消息和运行库里的消息都会使用正确的语言。缺点是你必须在启动 Vim 前设定所要用的语言。如果你想在运行 Vim 当中改变所用的语言，你就得使用第二种办法：

```
:language fr_FR.ISO_8859-1
```

这样你可以尝试使用几种不同的语种名。当你所指定的语言系统不支持时你会得到一个错误消息。而当所指定语言的消息文件不存在时，你不会被警告。Vim 只会悄悄的使用缺省的英语。

要找到系统支持的语言，先要找到列出这些语言的目录。在我的系统上，这个目录是 "/usr/share/locale"。有些系统上是 "/usr/lib/locale"。"setlocale" 的手册页应该告诉你该目录在你使用系统的什么地方。

在键入那些语种名的时候要当心，大小写是有区别的，而且 '-' 和 '_' 字符也很容易被混淆。

你也可以分别为消息，编辑文本和时间等设定所使用的语言。参阅 :language 。

自己动手翻译消息

如果翻译的消息没有包括你的语言，你也可以自己写。你得先拿到 Vim 的源代码和 GNU gettext 软件包。在解压缩了源代码后，先读一下 src/po/README.txt。那里有详细的指导。

翻译本身不是很难的，你不一定要是一个程序员。当然，你必须懂英语和你要翻译成的语言。

当你对你的翻译满意的时候，可以考虑和其它人共享。上载到 vim-online () 或者 e-mail 给 Vim 的维护者 Vim <maintainer@vim.org>。
或者两者都做。

45.2 菜单所用语言

缺省情况下菜单是英语的。要使用你自己的语言，首先它们得被翻译过。通常情况下菜单使用的语言会自动被设置成你所用的语言，就像消息一样。你什么也不用做。但是如果 Vim 不包括你所用语言的菜单的翻译就不行了。

假设你在德国，使用的语言被设为德语。但你更愿意在菜单中看到 "File" 而非 "Datei"。你可以这样将菜单语言换回英语：

```
:set langmenu=none
```

也可以指定某一语言：

```
:set langmenu=nl_NL.ISO_8859-1
```

和上面一样 "-" 和 "_" 是有区别的。然而，大小写在这里可以不分。

'langmenu' 选项必须在菜单被加载前设置才有效。一旦菜单已经被显示了，再改变其值就不会直接生效了。因此，你需要把设置 'langmenu' 的命令放在 vimrc 文件里。

如果你实在想在 Vim 运行当中改变菜单所用的语言，你可以这样做：

```
:source $VIMRUNTIME/delmenu.vim
:set langmenu=de_DE.ISO_8859-1
:source $VIMRUNTIME/menu.vim
```

这样有一个缺陷：所有你自己定义的菜单都不见了。你还得必须重新定义它们。

自己动手翻译菜单

在下面这个目录可以找到那些已有的菜单翻译：

```
$VIMRUNTIME/lang
```

翻译文件被称为 menu_{language}.vim。如果找不到你想使用的语言的翻译，你可以自行

翻译。最简单的办法就是拷贝现存的语言文件并在其基础上做改动。
先用 ":language" 命令找出你现在使用的语言的名称。就用这个名称，但名字的所有字母要小写。将这个文件拷贝到你自己的运行时目录。例如，在 Unix 系统上你应该：

```
:!cp $VIMRUNTIME/lang/menu_ko_kr.euckr.vim ~/.vim/lang/menu_nl_be.iso_8859-1.vim
```

你会在 "$VIMRUNTIME/lang/README.txt" 文件中找到与翻译相关的更多提示。

45.3 使用其它编码

Vim 猜你所要编辑的文件编码和你所用的语言相一致。对于许多欧洲语言而言，编码是 "latin1"。那么每个字节是一个字符。一共可能有 256 个不同的字符。而对于亚洲语言这就远远不够了。这些语言大多使用双字节编码，从而提供超过一万个可能的字符。即使如此，如果一篇文本中包括数种语言的文字，这样还是不够。Unicode 就是来解决这个问题的。其设计允许它包括多种语言中全部的常用字符。它被称为 "替代所有其它编码的超级编码格式"。但它的使用还不很广泛。
幸运的是，Vim 对这三种编码方式都支持，并且有条件的支持使用与系统环境语言不一致的文本编码。
当你的系统语言与你所编辑文件用的编码一致时，缺省设定应该没有问题。所以你什么也不用担心。下面的叙述只是当你想编辑不同的语言时才有关系。

在 GUI 上使用 UNICODE

Unicode 一个很方便的特性就是它可以与其它编码格式相会转换而不丢失信息。当你使 Vim 在内部使用 Unicode，你将可以用任何编码编辑文件。
不过，支持 Unicode 的系统还是比较有限的。因此可能你的语言不使用 Unicode。你得告诉 Vim 你希望使用 Unicode，然后在解决与系统其它部分交互的问题。
我们来启动 GUI 版本的 Vim，这样我们就可以显示 Unicode 字符。这样就行了：

```
:set encoding=utf-8
:set guifont=-misc-fixed-medium-r-normal--18-120-100-100-c-90-iso10646-1
```

'encoding' 选项告诉 Vim 你所用的字符的编码。该选项适用于缓冲区的文本（你正在编辑的文件），寄存器，Vim 脚本文件等等。你可以把 'encoding' 选项当作是对 Vim 内部运行机制的设定。
上例假设你的系统安装了指定的字体。例子中的字体名是用于 X-Window 系统的。这种字体是包括在一个用于增强 xterm 的软件包中的。如果你没有这种字体，你可以在这里找到：

对于 MS-Windows，一些字体包含了一些有限的 Unicode 字符。试一下 "Courier New" 字体。你可以通过 Edit/Select Font... 菜单来选择几种字体试试。但是你只能使用定宽的字体。例如：

```
:set guifont=courier_new:h12
```

如果不行的话，试着找个字体包。你可以在下面这个地址找到，如果 Microsoft 没把它移走的话（译者注：已经不能用了）：

现在你已经通知了 Vim 在内部使用 Unicode 并用一种 Unicode 字体显示文本。然而，Vim 接收到的输入字符依然来自原来语言的编码字符。这些字符必须被转换成 Unicode。你可以通过 'termencoding' 选项值来通知 Vim 要转换的语言。像这样：

```
:let &termencoding = &encoding
:set encoding=utf-8
```

这样就在 'encoding' 赋值为 utf-8 之前把 'encoding' 的旧值赋给 'termencoding'。你得自己试试这个办法在你的系统上行不行得通。当你使用一个为亚洲语言专门设计的输入法，同时又想编辑 Unicode 文本，这个办法应该工作的很好。

在 UNICODE 终 端 内 使 用 UNICODE

也有直接支持 Unicode 的终端。XFree86 自带的标准 xterm 就是其中之一。我们就用它作为例子。

首先，Unicode 支持要被编译进 xterm。参阅 UTF8-xterm 。

用 "-u8" 参数启动 xterm。你可能还需要指定一个字体。例如：

```
xterm -u8 -fn -misc-fixed-medium-r-normal--18-120-100-100-c-90-iso10646-1
```

现在你就可以在终端内运行 Vim 了。像前面一样将 'encoding' 设定为 "utf-8"。这样就行了。

在 普 通 终 端 里 使 用 UNICODE

假定你希望编辑 Unicode 文件，但是手头又没有一个支持 Unicode 的终端。你还是可以用 Vim 这样做，但是那些终端不支持的字符就不能被正确的显示了。文本的编排格式会被保留。

```
:let &termencoding = &encoding
:set encoding=utf-8
```

这和 GUI 的那个例子一样。但内部工作机制不同：Vim 会把显示的字符转换后再送给终端程序。这样屏幕就不会显示一堆毫无意义的字符。

这需要 'termencoding' 和 'encoding' 之间的转换。Vim 会自己处理 latin1 和 Unicode 之间的转化。其它的转换需要编译进来 +iconv 特性。

试着编辑一个含有 Unicode 的文件。你会 注意 到 Vim 会在那些无法显示的字符处显示问号（或下划线或者其它什么）。将光标移动到问号的地方并使用下面的命令：

```
ga
```

Vim 会显示该字符的代码。这样你就会知道那个字符大概是什么。你可以去查 Unicode 表。如果你有很多很多时间的话，你甚至可以这样来阅读一个文件的内容。

> 备注：
> 因为 Vim 对所有的文本都使用 'encoding' 选项。改变该选项的值会使所有的非 ASCII 码字符无效。你在使用寄存器和 'viminfo' 文件（例如，一个被存储的查找模式）时就会 注意 到了。建议在 vimrc 文件里设置 'encoding' 选项，然后就别再碰它。

45.4 编辑其它编码的文件

假定你已经配置 Vim 使用 Unicode 了，你现在想编辑一个 16 位 Unicode 的文件。听起来很简单，是吗？事实上，Vim 在内部使用 utf-8 编码，所以 16 为的编码必须被转换。因为存在一个字符集 (Unicode) 和编码格式 (utf-8 或 16 位) 的差别。

Vim 会试着用 'fileencodings' 选项中的编码名称检测你在编辑哪种文件。当使用 Unicode 时，缺省的选项值是："ucs-bom,utf-8,latin1"。也就是说 Vim 会检查编辑的文件看看是下面的编码之一：

ucs-bom	文件必须以 Byte Order Mark (BOM) 开始。可以检测 16 位，32 位和 utf-8 Unicode 编码。
utf-8	utf-8 Unicode。如果有在 utf-8 中非法的字节序列，此设定将被拒绝。
latin1	保证有效的经典 8 位编码。

当你开始编辑一个 16 位的 Unicode 编码，而该文件有 BOM 时，Vim 会检测到这些并在读取文件时将其转换为 utf-8 编码。'fileencoding' 选项 (后面没有 s) 被设为检测到的值，在本情况下是 "utf-16le"。表示是 Unicode，双字节以及 little-endian (高位字节在后) 的编码。这样的文件格式在 MS-Windows 上很常用 (例如，注册表文件)。

当写入文件时，Vim 会比较 'fileencoding' 和 'encoding' 的值。如果它们的值不同，文本会被转换。

如果 'fileencoding' 的值是空的，意味着不作任何转换。这样文本就会被以 'encoding' 的值来编码。

如果缺省的 'fileencodings' 值不行的话，你可以将其设成你希望 Vim 尝试使用的编码。只有当一个值无效时下一个才会被使用。将 "latin1" 放到第一位是不行的，因为它永远都不会是无效的。举个例子，要在文件没有 BOM 也不是 utf-8 编码的时候缺省设为简体中文编码 (Unix 和 Windows 上都可以用 chinese 作为编码名)：

```
:set fileencodings=ucs-bom,utf-8,chinese
```

encoding-values 有建议值。其它的值也可能有效。这取决于系统提供的转换功能。

强 制 编 码

如果编码的自动检测对你不起作用的话，你就得告诉 Vim 你要编辑的文件是什么编码。例如：

```
:edit ++enc=koi8-r russian.txt
```

"++enc" 部分指定编辑该文件时应该使用的编码。Vim 会把指定的编码 (在这是俄语) 转换为 'encoding'。'fileencoding' 也会被设为指定的编码，这样当写入文件时就会发生编码的反转换。

在写入文件时也可以用到一样的参数。这样你事实上可以用 Vim 来转换一个文件。例如：

```
:write ++enc=utf-8 russian.txt
```

备注：
转换可能导致字符丢失。在 Unicode 和其他编码之间的相互转换基本上不会有这个麻烦，除非你有非法字符。当文件包括多于一种语言的字符时，从 Unicode 转换为其它编码很可能会丢失信息。

45.5 文本录入

计算机键盘只有那么多键。有些语言包括几千个字符。Unicode 有超过十万。那你怎么才能输入这些字符呢?

首先，如果你不需要很多的特殊字符，你可以用二合字母。这已经在 24.9 讲述过了。

当你使用一种有很多字符的语言，你的键盘应付不了，你就得使用一种输入法 (IM)。这需要学习如何把输入的键变成要输入的字符。你所用的系统很可能已经提供了输入法。Vim 应该可以很好地与其合作。更多的细节请参阅 mbyte-XIM (X-Windows) 和 mbyte-IME (MS-Windows)。

键 盘 映 射 表

对于某些语言来说字符集可能不同但是可用的字符数目和拉丁字符集差不多。这样就有可能在字符和键盘之间定义一个一对一的对应。Vim 用键盘映射表来解决这个问题。

假定你想输入希伯来语，你可以这样载入希伯来语的键盘映射表:

```
:set keymap=hebrew
```

Vim 会为你找到一个键盘映射表文件。这取决于 'encoding' 选项的值。如果找不到相应的文件，你会得到一个错误信息。

现在你就可以在插入方式下输入希伯来语了。普通模式下，当你键入 ":"，Vim 会自动切换到英语。你也可以用下面这个命令来在希伯来语和英语之间切换:

```
CTRL-^
```

它只在插入模式和命令行模式下才有效。在普通模式下会产生完全不同的效果 (跳转到轮换文件)。

如果你打开 'showmode' 选项的话，键盘映射表的使用情况会在状态信息里指出。在 GUI 上，Vim 会使用不同的光标颜色来指明映射表的使用情况。

你可以用 'iminsert' 和 'imsearch' 选项来改变键盘映射表的用法。

要查看键映射的列表，用这个命令:

```
:lmap
```

要查看系统有哪些键盘映射表文件，在 GUI 上你可以用 Edit/Keymap 菜单。否则你可以用:

```
:echo globpath(&rtp, "keymap/*.vim")
```

(译者注: 中文用户可以试试 pinyin 映射表，可用来输入带四声的韵母。)

自 定 义 键 盘 映 射 表

你也可以创建你自己的键盘映射表。不是很难。从一个和你要使用语言类似的映射表开始。将其拷贝到你的运行时目录里的 "keymap" 子目录。例如，在 Unix 上，你可以用 "~/.vim/keymap" 目录。

键盘映射表的文件名必须遵守下面的规则:

```
keymap/{name}.vim
```

或

```
keymap/{name}_{encoding}.vim
```

{name} 是键盘映射表的名字。选择一个容易识别，又和系统中其它映射表不同的名字(除非你希望替换一个现存的映射表)。名字不能包括下划线。使用的编码名是可选的。例如:

```
keymap/hebrew.vim
keymap/hebrew_utf-8.vim
```

文件的内容应该很容易理解。看看随 Vim 一起发布的几个键盘映射表就行了。更详细的，请参阅 mbyte-keymap 。

最 后 的 办 法

如果其它的方法都行不通，你可以用 CTRL-V 来输入任何字符:

编码	键入	范围
8-bit	CTRL-V 123	十进制 0-255
8-bit	CTRL-V x a1	十六进制 00-ff
16-bit	CTRL-V u 013b	十六进制 0000-ffff
31-bit	CTRL-V U 001303a4	十六进制 00000000-7fffffff

注意 不要输入空格。更详细的，参阅 i_CTRL-V_digit 。

下一章: usr_50.txt 高级 Vim 脚本编写

VIM 用户手册 - by Bram Moolenaar
译者：Willis

高级 Vim 脚本编写

50.1 例外

让我们从一个例子开始：

```
try
   read ~/templates/pascal.tmpl
catch /E484:/
   echo "Sorry, the Pascal template file cannot be found."
endtry
```

如果该文件不存在的话， read 命令就会失败。这段代码可以捕捉到该错误并向用户给出一个友好的信息，而不是一个一般的出错信息。

在 try 和 endtry 之间的命令产生的错误将被转变成为例外。例外以字符串的形式出现。当错误发生时，该字符串包含出错信息。而每一个出错信息都有一个对应的错误码。在上面的例子中，我们捕捉到的错误包括 "E484:"。Vim 确保这个错误码始终不变(文字可能会变，例如被翻译)。

除了使出错信息更友好之外，Vim 也会继续执行 :endtry 之后的命令。否则，一旦遇到未捕获的错误，脚本/函数/映射的执行会立即中止。

当 read 命令引起其它错误时，模式 "E484:" 不会被匹配。因此该例外不会被捕获，结果是一个一般的出错信息而且执行被中止。

你可能想这样做：

```
try
   read ~/templates/pascal.tmpl
catch
   echo "Sorry, the Pascal template file cannot be found."
endtry
```

这意味着所有的错误都将被捕获。然而这样你就完全无法得到那些指出不同问题的错误信息，比如说 "E21: Cannot make changes, 'modifiable' is off"。捕获所有错误请三思而后行！

另一个有用的机制是 finally 命令:

```
var tmp = tempname()
try
   exe ":.,$write " .. tmp
   exe "!filter " .. tmp
   :.,$delete
   exe ":$read " .. tmp
finally
   delete(tmp)
endtry
```

这个例子将自光标处到文件尾的所有行通过过滤器 "filter"。该程序的参数是文件名。无论过滤器是否正常工作、 try 和 finally 之间发生了什么错误、或者用户按 CTRL-C 中断了过滤器， delete(tmp) 命令始终被执行。这可以确保你在任何情况下不会留下一个临时文件。

finally 本身并不捕获例外，错误仍然会中止后续行的执行。

关于例外处理更多的讨论可以阅读参考手册： exception-handling 。

50.2 带可变数目参数的函数

Vim 允许你定义参数个数可变的函数。下面的例子给出一个至少有一个参数 (start)，但可以多达 20 个附加参数的函数:

```
def Show(start: string, ...items: list<string>)
```

函数中的变量 "items" 会是包含额外参数的列表。用法就像普通的列表，如:

```
def Show(start: string, ...items: list<string>)
  echohl Title
  echo "start is " .. start
  echohl None
  for index in range(len(items))
    echon $"  Arg {index} is {items[index]}"
  endfor
  echo
enddef
```

可以这样调用:

```
Show('Title', 'one', 'two', 'three')
start is Title  Arg 0 is one  Arg 1 is two  Arg 2 is three
```

上例中 echohl 命令被用来给出接下来的 echo 命令如何高亮输出。`echohl None` 终止高亮。 echon 命令除了不输出换行符外，和 echo 一样。

如果调用时只给出一个参数，"items" 列表会为空。
range(len(items)) 返回索引的列表，可以在其上用 for 循环，这方面后面会继续解释。

50.3 恢复视窗

有时你想跳转到别处，作些修改，然后跳回到原先的位置和视图。例如要修改文件头部的一部分。可用两个函数实现：

```
var view = winsaveview()
# 移动并作修改
winrestview(view)
```

下一章： usr_51.txt 创建插件

usr_51.txt　　适用于 Vim 9.1 版本。　　最近更新：2025年3月

VIM 用户手册 - by Bram Moolenaar
译者：Willis

编写插件

插件可用于定义指定类型文件的设置，语法高亮和许多其它特性。本章解释如何编写最常见的 Vim 插件。

下一章：　usr_52.txt　　编写大型插件
前一章：　usr_50.txt　　高级 Vim 脚本编写
目录：　usr_toc.txt

51.1　编写通用插件　　write-plugin

用约定方式编写的脚本能够被除作者外的很多人使用。这样的脚本叫做插件。Vim 用户只要把你写的脚本放在 plugin 目录下就可以立即使用了：　add-plugin 。

实际上有两种插件：

全局插件：适用于所有类型的文件。
文件类型插件：仅适用于某种类型的文件。

这一节将介绍第一种。很多的东西也同样适用于编写文件类型插件。仅适用于编写文件类型插件的知识将在下一节　write-filetype-plugin　做介绍。

这里我们用　Vim9　语法，编写新插件的推荐方式。请确保文件以　vim9script　命令开始。

插 件 名

首先你得给你的插件起个名字。这个名字应该很清楚地表示该插件的用途。同时应该避免同别的插件用同样的名字而用途不同。

一个纠正打字错误的插件可能被命名为 "typecorrect.vim"。我们将用这个名字来举例。

为了使一个插件能被所有人使用，要 注意 一些事项。下面我们将一步步的讲解。最后会给出这个插件的完整示例。

插 件 体

让我们从做实际工作的插件体开始：

```
12      iabbrev teh the
13      iabbrev otehr other
14      iabbrev wnat want
15      iabbrev synchronisation
16              \ synchronization
```

当然，真正的清单会比这长的多。

上面的行号只是为了方便解释，不要把它们也加入到你的插件文件中去！

首 行

```
 1      vim9script noclear
```

vim9script 必须是第一个命令，建议放在文件的首行上。

我们编写的脚本会有 finish 命令在第二次载入时立即退出。使用 "noclear" 参数可以避免脚本定义的项目丢失。更多细节可见 vim9-reload 。

插 件 头

你很可能对这个插件做新的修改并很快就有了好几个版本。并且当你发布文件的时候，别人也想知道是谁编写了这样好的插件或者给作者提点意见。所以，在你的插件头部加上一些描述性的注释是很必要的：

```
 2      # Vim global plugin for correcting typing mistakes
 3      # Last Change:  2021 Dec 30
 4      # Maintainer:   Bram Moolenaar <Bram@vim.org>
```

关于版权和许可：由于插件很有用，而且几乎不值得限制其发行，请考虑对你的插件使用公共领域 (public domain) 或 Vim 许可 license 。在文件顶部放上说明就行了。例如：

```
 5      # License:      This file is placed in the public domain.
```

禁 止 加 载

有可能一个用户并不总希望加载这个插件。或者系统管理员在系统的插件目录中已经把这个插件删除了，而用户希望使用它自己安装的插件。用户应该有机会选择不加载指定的插件。下面的一段代码就是用来实现这个目的的：

```
 7      if exists("g:loaded_typecorrect")
 8        finish
 9      endif
10      g:loaded_typecorrect = 1
```

这同时也避免了同一个脚本被加载两次以上。因为那样函数会无意义地被重新定义，自动命令被多次加入也会引起问题。

建议使用的名字以 "g:loaded_" 开头，然后是插件的文件名，按原义输入。之前加上

"g:" 使此变量成为全局的，这样其它地方可以检查插件功能是否已存在。如果没有 "g:" 这将是局部于函数的变量。

finish 阻止 Vim 继续读入文件的其余部分，这比用 if-endif 包围整个文件要快得多，因为 Vim 会需要解析所有这些命令以找到 endif 。

映 射

现在让我们把这个插件变得更有趣些：我们将加入一个映射用来校正当前光标下的单词。我们当然可以任意选一个键组合，但是用户可能已经将其定义为其它的什么功能了。为了使用户能够自己定义在插件中的键盘映射使用的键，我们可以使用 <Leader> 标识：

```
20       map <unique> <Leader>a  <Plug>TypecorrAdd;
```

那个 "<Plug>TypecorrAdd;" 会做实际的工作，后面我们还会做更多解释。

用户可以将 "g:mapleader" 变量设为用户所希望的开始插件映射的键组合。比如假设用户这样做：

```
        g:mapleader = "_"
```

映射将定义为 "_a"。如果用户没有这样做，Vim 将使用缺省值反斜杠。这样就会定义一个映射 - "\a"。

注意 其中用到了 <unique>，这会使得 Vim 在映射已经存在时给出错误信息。
:map-<unique>

但是如果用户希望定义自己的键操作呢？我们可以用下面的方法来解决：

```
19     if !hasmapto('<Plug>TypecorrAdd;')
20       map <unique> <Leader>a  <Plug>TypecorrAdd;
21     endif
```

我们先检查对 "<Plug>TypecorrAdd;" 的映射是否存在。仅当不存在时我们才定义映射 "<Leader>a"。这样用户就可以在用户自己的 vimrc 文件中加入：

```
        map ,c  <Plug>TypecorrAdd;
```

那么键序列就会是 ",c" 而不是 "_a" 或者 "\a" 了。

分 割

如果一个脚本变得相当长，你通常希望将其分割成几部分。常见做法是函数或映射。但同时，你又不希望脚本之间这些函数或映射相互干扰。例如，你定义了一个函数 Add()，但另一个脚本可能也试图定一同名的函数。为了避免这样的情况发生，我们要使函数局部于脚本。幸运地是， Vim9 脚本里这已经是缺省。在老式脚本里需要在名字的前面加上 "s:"。

我们来定义一个用来添加新的错误更正的函数：

```
28     def Add(from: string, correct: bool)
29       var to = input($"type the correction for {from}: ")
```

```
30        exe $":iabbrev {from} {to}"
...
34      enddef
```

这样我们就可以在这个脚本之内调用函数 Add()。如果另一个脚本也定义 Add()，该函数将只能在其所定义的脚本内部被调用。独立于这两个函数的全局的 g:Add() 函数也可以存在。

映射则可用 <SID>。它产生一个脚本 ID。在我们的错误更正插件中我们可以做以下的定义：

```
22      noremap <unique> <script> <Plug>TypecorrAdd;  <SID>Add
...
26      noremap <SID>Add  :call <SID>Add(expand("<cword>"), true)<CR>
```

这样当用户键入 "\a" 时，将触发下面的次序：

```
        \a  ->  <Plug>TypecorrAdd;  ->  <SID>Add  ->  :call <SID>Add(...)
```

如果另一个脚本也定义了映射 <SID>Add，该脚本将产生另一个脚本 ID。所以它定义的映射也与前面定义的不同。

注意 在这里我们用了 <SID>Add() 而不是 Add()。这是因为该映射将被用户键入，因此是从脚本上下文的外部调用的。<SID> 被翻译成该脚本的 ID。这样 Vim 就知道在哪个脚本里寻找相应的 Add() 函数了。

这的确是有点复杂，但又是使多个插件一起工作所必需的。基本规则是：在映射中使用 <SID>Add()；在其它地方（该脚本内部，自动命令，用户命令）使用 Add()。

我们还可以增加菜单项目来完成映射同样的功能：

```
24      noremenu <script> Plugin.Add\ Correction      <SID>Add
```

建议把插件定义的菜单项都加入到 "Plugin" 菜单下。上面的情况只定义了一个菜单选项。当有多个选项时，可以创建一个子菜单。例如，一个提供 CVS 操作的插件可以添加 "Plugin.CVS" 子菜单，并在其中定义 "Plugin.CVS.checkin"，"Plugin.CVS.checkout" 等菜单项。

注意 为了避免其它映射引起麻烦，在第 28 行使用了 ":noremap"。比如有人可能重新映射了 ":call"。在第 24 也用到了 ":noremap"，但我们又希望重新映射 "<SID>Add"。这就是为什么在这儿要用 "<script>"。它允许只执行局部于脚本的映射。
:map-<script> 同样的道理也适用于第 26 行的 ":noremenu"。 :menu-<script>

<SID> 和 <Plug> **using-<Plug>**

<SID> 和 <Plug> 都是用来避免映射的键序列和那些仅仅用于其它映射的映射起冲突。
注意 <SID> 和 <Plug> 的区别：

<Plug> 在脚本外部是可见的。它被用来定义那些用户可能定义映射的映射。<Plug> 是无法用键盘输入的特殊代码。
使用结构：<Plug> 脚本名 映射名，可以使得其它插件使用同样次序的字符来定义映射的几率变得非常小。

在我们上面的例子中，脚本名是 "Typecorr"，映射名是 "Add"。我们加上一个分号作为终止符。结果是 "<Plug>TypecorrAdd;"。只有脚本名和映射名的第一个字符是大写的，所以我们可以清楚地看到映射名从什么地方开始。

<SID>　是脚本的 ID，用来唯一的代表一个脚本。Vim 在内部将 <SID> 翻译为 "<SNR>123_"，其中 "123" 可以是任何数字。这样一个函数 "<SID>Add()" 可能在一个脚本中被命名为 "<SNR>11_Add()"，而在另一个脚本中被命名为 "<SNR>22_Add()"。如果你用 ":function" 命令来获得系统中的函数列表你就可以看到了。映射中对 <SID> 的翻译是完全一样的。这样你才有可能通过一个映射来调用某个脚本中的局部函数。

用 户 命 令

现在让我们来定义一个用来添加更正的用户命令：

```
36     if !exists(":Correct")
37       command -nargs=1  Correct  :call Add(<q-args>, false)
38     endif
```

这个用户命令只在系统中没有同样名称的命令时才被定义。否则我们会得到一个错误。用 ":command!" 来覆盖现存的用户命令是个坏主意。这很可能使用户不明白自己定义的命令为什么不起作用。 :command
如果确实发生了，可以找到谁该被骂：

```
verbose command Correct
```

脚 本 变 量

当一个变量前面带有 "s:" 时，我们将它称为脚本变量。该变量只能在脚本内部被使用。在脚本以外该变量是不可见的。这样就避免了在不同的脚本中使用同一个变量名的麻烦。该变量在 Vim 的运行期间都是可用的。当再次调用 (source) 该脚本时使用的是同一个变量。 s:var

Vim9　脚本的好处是变量缺省都是脚本局部的。可以加 "s" 前缀，但没必要。脚本里的函数也可直接用脚本变量而无需前缀 (为此，必须在函数前先声明)。

脚本局部变量也可以在脚本定义的函数、自动命令和用户命令中使用。所以它们是插件各部分共享信息而不用担心外泄的最佳途径。在我们的例子中我们可以加入几行用来统计更正的个数：

```
17     var count = 4
...
28     def Add(from: string, correct: bool)
...
32       count += 1
33       echo "you now have " .. count .. " corrections"
34     enddef
```

起初 "count" 被脚本声明以及初始化为 4。当后来 Add() 函数被调用时，其值被增加了。在哪里调用函数无关紧要。只要它是定义在该脚本以内的，就可以使用脚本中的局部变量。

结 果

下面就是完整的例子:

```
 1     vim9script noclear
 2     # Vim global plugin for correcting typing mistakes
 3     # Last Change:  2021 Dec 30
 4     # Maintainer:   Bram Moolenaar <Bram@vim.org>
 5     # License:      This file is placed in the public domain.
 6
 7     if exists("g:loaded_typecorrect")
 8       finish
 9     endif
10     g:loaded_typecorrect = 1
11
12     iabbrev teh the
13     iabbrev otehr other
14     iabbrev wnat want
15     iabbrev synchronisation
16             \ synchronization
17     var count = 4
18
19     if !hasmapto('<Plug>TypecorrAdd;')
20       map <unique> <Leader>a  <Plug>TypecorrAdd;
21     endif
22     noremap <unique> <script> <Plug>TypecorrAdd;  <SID>Add
23
24     noremenu <script> Plugin.Add\ Correction      <SID>Add
25
26     noremap <SID>Add  :call <SID>Add(expand("<cword>"), true)<CR>
27
28     def Add(from: string, correct: bool)
29       var to = input("type the correction for " .. from .. ": ")
30       exe ":iabbrev " .. from .. " " .. to
31       if correct | exe "normal viws\<C-R>\" \b\e" | endif
32       count += 1
33       echo "you now have " .. count .. " corrections"
34     enddef
35
36     if !exists(":Correct")
37       command -nargs=1  Correct  call Add(<q-args>, false)
38     endif
```

第 31 行还没有解释过。它将新定义的更正用在当前光标下的单词上。 :normal 被用来使用新的缩写。 注意 虽然这个函数是被一个以 ":noremap" 定义的映射调用的，这里的映射和缩写还是被展开使用了。

文 档 write-local-help

给你的插件写一些文档是个好主意。特别是当用户可以自定义其中的某些功能时尤为必要。关于帮助文件使用的语法，可查阅 help-writing ，而关于如何安装本地帮助文档，请查阅 add-local-help 。

下面是一个插件帮助文档的简单例子，名叫 "typecorrect.txt":

```
 1  *typecorrect.txt*       Plugin for correcting typing mistakes
 2
 3  If you make typing mistakes, this plugin will have them corrected
 4  automatically.
 5
 6  There are currently only a few corrections.  Add your own if you like.
 7
 8  Mappings:
 9  <Leader>a   or   <Plug>TypecorrAdd;
10          Add a correction for the word under the cursor.
11
12  Commands:
13  :Correct {word}
14          Add a correction for {word}.
15
16                                                  *typecorrect-settings*
17  This plugin doesn't have any settings.
```

其实只有第一行是文档的格式所必需的。Vim 将从该帮助文件中提取该行并加入到 help.txt 的 "LOCAL ADDITIONS:" local-additions （本地附加文档）一节中。第一个 "*" 一定要在第一行的第一列。加入你的帮助文件之后用 ":help" 来检查一下各项是否很好的对齐了。

你可以为你的帮助文档在 ** 之间加入更多的标签。但请注意不要使用现存的帮助标签。你最好能在标签内使用插件名用以区别，比如上例中的 "typecorrect-settings"。

建议使用 || 来引用帮助系统中的其它部分。这可以使用户很容易得找到相关联的帮助。

小 结 plugin-special

关于插件的小结:

var name	脚本的局部变量。
<SID>	脚本 ID，用于局部于脚本的映射和函数。
hasmapto()	用来检测插件定义的映射是否已经存在的函数。
<Leader>	"mapleader" 的值。用户可以通过该变量定义插件所定义映射的起始字符。
map <unique>	如果一个映射已经存在的话，给出警告信息。
noremap <script>	在映射右边仅执行脚本的局部映射，而不检查全局映射。
exists(":Cmd")	检查一个用户命令是否存在。

51.2 编写文件类型插件 write-filetype-plugin ftplugin

文件类型插件和全局插件其实很相似。但是它的选项设置和映射等仅对当前缓冲区有效。这类插件的用法请参阅 add-filetype-plugin 。

请先阅读上面 51.1 关于全局插件的叙述。其中所讲的对文件类型插件也都适用。这里只讲述一些不同之处。最根本的区别是文件类型插件只应该对当前缓冲区生效。

禁 用

如果你在编写一个提供很多人使用的文件类型插件，这些用户应该有机会选择不加载该插件。你应该在插件的顶端加上:

```
# Only do this when not done yet for this buffer
if exists("b:did_ftplugin")
  finish
endif
b:did_ftplugin = 1
```

这同时也避免了同一插件在同一缓冲区内被多次执行的错误（当使用不带参数的 ":edit" 命令时就会发生)。

现在用户只要编写一个如下几行的文件类型插件就可以完全避免加载缺省的文件类型插件了:

```
vim9script
b:did_ftplugin = 1
```

当然这要求该文件类型插件所处的文件类型插件目录在 'runtimepath' 所处的位置在 $VIMRUNTIME 之前!

如果你的确希望使用缺省的插件，但是又想自行支配其中的某一选项，你可以用一个类似下例的插件:

```
set textwidth=70
```

现在将这个文件存入那个 "after" 目录中。这样它就会在调用 Vim 本身的 "vim.vim" 文件类型插件之后被调用 after-directory 。对于 Unix 系统而言，该目录会是 "~/.vim/after/ftplugin/vim.vim"。 注意 缺省的插件设置了 "b:did_ftplugin"，而此脚本忽略之。

选 项

为了确保文件类型插件仅仅影响当前缓冲区，应该使用

```
setlocal
```

命令来设置选项。还要 注意 只设定缓冲区的局部选项（查查有关选项的帮助)。当 :setlocal 被用于设置全局选项或者某窗口的局部选项时，会影响到多个缓冲区，这是文件类型插件应该避免的。

当一个选项的值是多个标志位或项目的 "合" 时，考虑使用 "+=" 和 "-=", 这样可以保留现有的值。 注意 用户可能已经改变了该选项的值了。通常先将选项的值复位成缺省值

再做改动是个好主意。例:

```
setlocal formatoptions& formatoptions+=ro
```

映 射

为了确保键盘映射只对当前缓冲区生效，应该使用

```
map <buffer>
```

命令。这还应该和上面讲述的两步映射法连起来使用。下面是一个例子:

```
if !hasmapto('<Plug>JavaImport;')
  map <buffer> <unique> <LocalLeader>i <Plug>JavaImport;
endif
noremap <buffer> <unique> <Plug>JavaImport; oimport ""<Left><Esc>
```

hasmapto() 被用来检查用户是否已经定义了一个对 <Plug>JavaImport; 的映射。如果没有，该文件类型插件就定义缺省的映射。因为缺省映射是以 <LocalLeader> 开始，就使得用户可以自己定义映射的起始字符。缺省的是反斜杠。
"<unique>" 的用途是当已经存在的了这样的映射或者和已经存在的映射有重叠的时候给出错误信息。
:noremap 被用来防止其他用户定义的映射干扰。不过，":noremap <script>" 仍然可以允许进行脚本中以 <SID> 开头的映射。

一定要给用户保留禁止一个文件类型插件内的映射而不影响其它功能的能力。下面通过一个邮件文件类型插件来演示如何做到这一点:

```
# 增加映射，除非用户反对。
if !exists("g:no_plugin_maps") && !exists("g:no_mail_maps")
  # Quote text by inserting "> "
  if !hasmapto('<Plug>MailQuote;')
    vmap <buffer> <LocalLeader>q <Plug>MailQuote;
    nmap <buffer> <LocalLeader>q <Plug>MailQuote;
  endif
  vnoremap <buffer> <Plug>MailQuote; :s/^/> /<CR>
  nnoremap <buffer> <Plug>MailQuote; :.,$s/^/> /<CR>
endif
```

其中用到了两个全局变量:

```
g:no_plugin_maps        禁止所有文件类型插件中的映射
g:no_mail_maps          禁止 "mail" 文件类型插件的映射
```

用 户 命 令

在使用 :command 命令时，如果加上 "-buffer" 开关，就可以为某一类型的文件加入一个用户命令，而该命令又只能用于一个缓冲区。例:

```
command -buffer  Make  make %:r.s
```

变 量

文件类型插件对每一个该类型的文件都会被调用。脚本局部变量会被所有的调用共享。如果你想定义一个仅对某个缓冲区生效的变量，使用缓冲区局部变量 b:var 。

函 数

一个函数只需要定义一次就行了。可是文件类型插件会在每次打开相应类型的文件时都被调用。下面的结构可以确保函数只被定义一次:

```
if !exists("*Func")
  def Func(arg)
    ...
  enddef
endif
```

不要忘记 vim9script 命令要用 "noclear"，以免脚本再次载入时删除函数。

撤 销 undo_indent undo_ftplugin

当用户执行 ":setfiletype xyz" 时，之前的文件类型命令应该被撤销。在你的文件类型插件中设定 b:undo_ftplugin 变量，用来撤销该插件的各种设置。例如:

```
b:undo_ftplugin = "setlocal fo< com< tw< commentstring<"
        \ .. "| unlet b:match_ignorecase b:match_words b:match_skip"
```

在 ":setlocal" 命令的选项后使用 "<" 会将其值复位为全局值。这可能是最好的复位选项值的方法。

要撤销缩进脚本的效果，必须相应地设定 b:undo_indent 变量。

这两个变量使用老式脚本语法，而不是 Vim9 语法。

文 件 名

文件类型必须被包括在插件文件名中 ftplugin-name 。可以使用以下三种形式之一:

```
.../ftplugin/stuff.vim
.../ftplugin/stuff_foo.vim
.../ftplugin/stuff/bar.vim
```

"stuff" 是文件类型，"foo" 和 "bar" 是任意名字。

文 件 类 型 检 测 plugin-filetype

如果 Vim 还不能检测到你的文件类型，你需要在单独的文件里创立一个文件类型检测的代码段。通常，它的形式是一个自动命令，它在文件名字匹配某模式时设置文件类型。例如:

```
au BufNewFile,BufRead *.foo                    set filetype=foofoo
```

把这个一行的文件写到 'runtimepath' 里第一个目录下的 "ftdetect/foofoo.vim"。Unix 上应该是 "~/.vim/ftdetect/foofoo.vim"。惯例是，使用文件类型的名字作为脚本的名字。

如果你愿意，你可以使用更复杂的检查。比如检查文件的内容以确定使用的语言。另见 new-filetype 。

小 结 ftplugin-special

以下是有关文件类型插件一些特殊环节：

<LocalLeader>	"maplocalleader" 的值，用户可以通过它来自定义文件类型插件中映射的起始字符。
map <buffer>	定义一个仅对缓冲区有效的局部映射。
noremap <script>	仅重映射脚本中以 <SID> 开始的映射。
setlocal	设定仅对当前缓冲区有效的选项。
command -buffer	定义一个仅对缓冲区有效的局部命令。
exists("*s:Func")	查看是否已经定义了某个函数。
参阅所有插件的特殊环节	plugin-special 。

51.3 编写编译器插件 write-compiler-plugin

编译器插件可以用来设定于某一特定编译器相关的选项。用户可以使用 :compiler 命令来加载之。主要是用以设定 'errorformat' 及 'makeprg' 选项。

最简单的方法就是学习一个例子。下面的命令将编辑所有缺省安装的编译器插件：

```
next $VIMRUNTIME/compiler/*.vim
```

用 :next 可以查阅下一个插件文件。

这类文件有两个特别之处。一是允许用户否决或者增强缺省文件的机制。缺省的文件以下面的代码开始：

```
vim9script
if exists("g:current_compiler")
  finish
endif
g:current_compiler = "mine"
```

当你写了编译器文件并把它放到你个人的运行时目录（例如，Unix 上 ~/.vim/compiler）时，你需要设置 "current_compiler" 变量，使得缺省文件不进行设置。

:CompilerSet

第二个特别之处是：用 ":set" 命令来配合 ":compiler!" 而用 ":setlocal" 来配合 ":compiler"。Vim 为此定义了 ":CompilerSet" 用户命令。下面是一个例子：

```
CompilerSet errorformat&            " use the default 'errorformat'
CompilerSet makeprg=nmake
```

注意：参数需要根据 option-backslach 转义。

当你为 Vim 发布版本或者整个系统编写编译器插件时，应该使用上面提到的机制。这样当用户插件已经定义了 "current_compiler" 的时候什么也不做。

当你为了自行定义缺省插件的一些设定而编写编译器插件时，不要检查 "current_compiler"。这个插件应该在最后加载，因此其所在目录应该在 'runtimepath' 的最后。对于 Unix 来说可能是 ~/.vim/after/compiler。

51.4 发布 Vim 脚本 distribute-script

Vim 用户可以在 Vim 网站上寻找脚本：。如果你实现了对别人也有用的功能，让大家一起分享！

另一个地方是 github。但那里你需要知道怎么去找。优点是绝大多数插件管理员都从 github 获取插件。用你喜欢的搜索引擎去找找吧。

Vim 脚本应该可以用于任何系统。不过，它们不一定有 tar 或 gzip 命令。如果你想把文件打包和/或进行压缩，建议使用 "zip" 工具。

最理想的可移植方法是让 Vim 自己给脚本打包，用 Vimball 工具。见 vimball 。

最好你能加入一行内容，实现自动更新。见 glvs-plugins 。

下一章： usr_52.txt 编写大型插件

usr_52.txt　　适用于 Vim 9.1 版本。　　最近更新：2025年3月

VIM 用户手册 - by Bram Moolenaar
译者：Willis

编写更大型插件

插件在开始完成更复杂的功能时，会越来越庞大。本文件说明如何确保仍然可以快速载入，和如何把大文件分割成小部分。

下一章：　usr_90.txt　　安装 Vim
前一章：　usr_51.txt　　创建插件
　目录：　usr_toc.txt

52.1　导出和导入

Vim9 脚本设计使大型 Vim 脚本的编写更为容易。它看来更像其它脚本语言，尤其是 Typescript。另外，函数被编译为指令以快速执行。这使 Vim9 脚本快很多，多达百倍。

这里的基本想法是脚本文件包含只用于脚本文件内部的私有项目，和可被脚本文件之外使用的导出项目。然后，这些导出项目可用于导入它们的脚本里。这样什么在哪里定义就会很清晰。

让我们从一个例子开始，有一个导出函数和有一个私有函数的脚本：

```
vim9script

export def GetMessage(count: string): string
   var nr = str2nr(count)
   var result = $'To {nr} we say '
   result ..= GetReply(nr)
   return result
enddef

def GetReply(nr: number): string
  if nr == 42
     return 'yes'
  elseif nr == 22
     return 'maybe'
  else
     return 'no'
  endif
enddef
```

vim9script 命令是必须的， export 只能用于 Vim9 脚本里。

`export def GetMessage(...` 行以 export 开始，意味着此函数可以被其它脚本调用。`def GetPart(...` 行不以 export 开头，意味着这是局部于脚本的函数，只能在此脚本文件中使用。

现在说说导入此脚本的脚本。此例中使用如下布局，这适用于放在 "pack" 目录之下的插件：

```
.../plugin/theplugin.vim
.../lib/getmessage.vim
```

假定 "..." 目录已添加进 'runtimepath'，Vim 会寻找 "plugin" 目录里的插件并载入 "theplugin.vim"。Vim 并不识别 "lib" 目录，你可以在那里放任何脚本。

上面导出 GetMessage() 的脚本放在 lib/getmessage.vim 里。GetMessage() 函数在 plugin/theplugin.vim 里调用：

```
vim9script

import "../lib/getmessage.vim"
command -nargs=1 ShowMessage echomsg getmessage.GetMessage(<f-args>)
```

import 命令使用了相对路径，因为它以 "../" 开始，也就是上一层目录。其它类型的路径可见 :import 命令。

现在试试插件提供的命令吧：

```
ShowMessage 1
To 1 we say no

ShowMessage 22
To 22 we say maybe
```

注意 GetMessage() 使用了导入脚本名字 "getmessage" 作为前缀。这样，对每个导入函数，你会知道它是从哪里导入的。如果导入多个脚本，每个脚本都可以定义自己的 GetMessage() 函数：

```
vim9script

import "../lib/getmessage.vim"
import "../lib/getother.vim"
command -nargs=1 ShowMessage echomsg getmessage.GetMessage(<f-args>)
command -nargs=1 ShowOther echomsg getother.GetMessage(<f-args>)
```

如果导入脚本名字太长或者需要在很多地方使用，可以通过 "as" 参数来缩短名字：

```
import "../lib/getmessage.vim" as msg
command -nargs=1 ShowMessage echomsg msg.GetMessage(<f-args>)
```

重 载

记住一件事：导入的 "lib/getmessage.vim" 脚本只会载入一次。第二次载入时会跳过，因为里面的项目都已经创建了。不管是在其它脚本还是同一脚本里再次执行导入命令，都是如此。

对插件的使用这是高效的，但在插件开发期间，这意味着导入 "lib/getmessage.vim" 之后对它的修改不会生效。为此，只能退出 Vim 后重启。(理据：脚本定义的项目可能在编译过的函数里使用，再次载入脚本可能会破坏这些函数)。

使 用 全 局 值

有时需要使用全局变量或函数，以便在任何地方都能使用。一个全局变量的好例子是控制插件行为的首选项。为了避免其它脚本使用的重名，使用别人不太可能使用的前缀。例如，如果插件是 "mytags" 的话：

```
g:mytags_location = '$HOME/project'
g:mytags_style = 'fast'
```

52.2 自动载入

在把大脚本分割为若干部分后，使用脚本时所有行仍然被载入和执行。每个 import 都会载入被导入的脚本以找到其中定义的项目。虽然这样容易早点发现错误，但颇耗时。如果提供的功能不常用，这很浪费。

import 除了立即载入脚本之外，也可以延迟载入。使用上面的例子，只要在 plugin/theplugin.vim 里改一处就可以了：

```
import autoload "../lib/getmessage.vim"
```

脚本其余部分无须改动。不过，不会检查类型。甚至 GetMessage() 函数存在与否，也是在实际使用时才会检查。需要你自己决定是快速启动还是早点发现错误更加重要。也可以在检查完一切都正常之后再加上 "autoload" 参数。

自 动 载 入 目 录

另一种使用自动载入的形式，脚本名既不是绝对也不是相对路径：

```
import autoload "monthlib.vim"
```

会在 'runtimepath' 的 autoload 目录下搜索脚本 "monthlib.vim"。Unix 一个常见目录是 "~/.vim/autoload"。也会在 'packpath' 里 "start" 之下搜索脚本。

这种形式的主要好处是容易和其它脚本共享。但要确保脚本名唯一，因为 Vim 会在 'runtimepath' 下搜索所有的 "autoload" 目录，如果使用插件管理器管理多个插件，它会在 'runtimepath' 下新增目录，每个目录都可能有 "autoload" 目录。

没有自动载入的话：

```
import "monthlib.vim"
```

Vim 会在 'runtimepath' 的 import 目录下搜索脚本 "monthlib.vim"。注意 这种形式里，加上或去掉 "autoload" 关键字会影响脚本寻找的位置。而使用相对或绝对路径的形式时位置是不会改变的。

52.3 不经导入/导出的自动载入

write-library-script

在导入/导出机制引入之前，已经有另一套有用的机制，有些用户会觉得它比较简单些。基本想法是调用一个有特殊名字的函数，然后这个函数在某自动载入脚本中。我们把这种脚本称为库脚本。

此自动载入机制是基于有 "#" 字符的函数名的：

```
mylib#myfunction(arg)
```

Vim 会识别有内嵌 "#" 字符的函数名，如果该函数还没有定义，查找 'runtimepath' 里的 "autoload/mylib.vim"。该脚本必须定义 "mylib#myfunction()" 函数。显然 "mylib" 名字是 "#" 之前的部分，加上 ".vim" 后就被用作脚本名。

在 mylib.vim 脚本里可以放上许多其它函数，你可以自由组织库脚本的函数。但必须使函数名 "#" 前面的部分匹配脚本名。否则 Vim 无法知道载入哪个脚本。这是和导入/导出机制不同之处。

如果你真的热情高涨写了很多库脚本，现在可能想要用子目录吧。例如：

```
netlib#ftp#read('somefile')
```

这里脚本名取自函数名最后 "#" 之前的部分。中间的 "#" 用斜杠替代，最后加上 ".vim"。这样得到的名字就是 "netlib/ftp.vim"。Unix 上，这里使用的库脚本可以是：

```
~/.vim/autoload/netlib/ftp.vim
```

其中的函数应该如此定义：

```
def netlib#ftp#read(fname: string)
        "  用 ftp 读入文件 fname
enddef
```

注意 定义所用的函数名必须和调用的函数名完全相同。最后一个 '#' 之前的部分必须准确匹配子目录和脚本名。

同样的机制可以用来定义变量：

```
var weekdays = dutch#weekdays
```

会载入脚本 "autoload/dutch.vim"，它应该包含这样的内容：

```
var dutch#weekdays = ['zondag', 'maandag', 'dinsdag', 'woensdag',
        \ 'donderdag', 'vrijdag', 'zaterdag']
```

进一步的阅读可见： autoload 。

52.4 可用的其它机制

有些人觉得维护多个文件很麻烦，而把所有内容放在一个脚本。为了避免这样引起的启动时的延迟，有一个机制只定义很小的一部分，而其余部分的载入会延迟到实际使用的时候。 write-plugin-quickload

基本的方法是调用插件两次。第一次定义用户命令和映射，提供需要的功能。第二次定义实现这些功能的函数。

听起来很吓人，快速载入意味着载入两次！我们的意思是，第一次载入很快，把脚本的大部分内容延迟到第二次才载入，只有实际使用这些功能时才会这么做。当然，如果你总是用这些功能，实际上更慢了！

这里使用 FuncUndefined 自动命令。和上述的 autoload 功能不同。

下例演示如何这是如何完成的：

```
" 演示快速载入的 Vim 全局插件
" Last Change:  2005 Feb 25
" Maintainer:   Bram Moolenaar <Bram@vim.org>
" License:      This file is placed in the public domain.

if !exists("s:did_load")
        command -nargs=* BNRead  call BufNetRead(<f-args>)
        map <F19> :call BufNetWrite('something')<CR>

        let s:did_load = 1
        exe 'au FuncUndefined BufNet* source ' .. expand('<sfile>')
        finish
endif

function BufNetRead(...)
        echo 'BufNetRead(' .. string(a:000) .. ')'
        " 读入功能在此
endfunction

function BufNetWrite(...)
        echo 'BufNetWrite(' .. string(a:000) .. ')'
        " 写回功能在此
endfunction
```

第一次载入脚本时，没有设置 "s:did_load"。这时执行 "if" 和 "endif" 之间的命令。它以 :finish 命令结束，这样脚本的其余部分不再执行。

第二次载入脚本时，"s:did_load" 已经存在，这时执行 "endif" 之后的命令。这里定义 (可能很长的) BufNetRead() 和 BufNetWrite() 函数。

如果把该脚本放到你的 plugin 目录，Vim 启动时会执行它。下面列出发生事件的序列：

1. 启动期间执行脚本时，定义 "BNRead" 命令并映射 <F19> 键。定义 FuncUndefined 自动命令。":finish" 命令使脚本提前终止。

2. 用户输入 BNRead 命令或者按了 <F19> 键。BufNetRead() 或 BufNetWrite() 函数会被调用。

3. Vim 找不到这些函数并因此激活了 FuncUndefined 自动命令事件。因为模式 "BufNet*" 匹配要调用的函数，执行命令 "source fname"，其中 "fname" 被赋予脚本的名字，不管它实际在何处都没问题。这是因为该名字来自 "<sfile>" 扩展的结果 (见 expand())。

4. 再次执行脚本。"s:did_load" 变量已经存在，此时定义函数。

注意 后来载入的函数匹配 FuncUndefined 自动命令的模式。要确信其它插件没有定义匹配此模式的函数。

52.5 在老式脚本中使用 Vim9 脚本 source-vim9-script

有些情况下，在老式脚本里你会想使用 Vim9 脚本里的项目。例如在 .vimrc 里初始化插件。最好的方法是用 :import 。例如:

```
import 'myNicePlugin.vim'
call myNicePlugin.NiceInit('today')
```

这会找到 Vim9 脚本文件里的 "NiceInit" 导出函数，并使之成为局部于脚本的项目 "myNicePlugin.NiceInit"。即使不给出 "s:"， :import 也总使用脚本命名空间，如果 "myNicePlugin.vim" 已经执行过，不会再次执行。

除了避免把任何项目放进全局命名空间（因为命名冲突会导致意料之外的错误）以外，这也意味着无论其中的项目被导入多少次，一份脚本只执行一次。

有些情况下，例如为了测试用途，可能就想要直接执行 Vim9 脚本。这没问题，但你只能得到其中的全局项目。该 Vim9 脚本要确保这些全局项目使用独一无二的名字。例如:

```
source ~/.vim/extra/myNicePlugin.vim
call g:NicePluginTest()
```

52.6 Vim9 examples: comment and highlight-yank plugin

注 释 包

Vim 带有一个 comment 插件，用 Vim9 脚本编写。 comment-install
看一下在 $VIMRUNTIME/pack/dist/opt/comment/ 的包的内容吧。

高 亮 抽 出 插 件

这里是一个高亮抽出区域的例子。使用 Vim 9.1.0446 开始可用的 getregionpos() 函数。

复制下例到一个新文件，把它放在你的 plugin 目录，下次你启动 Vim 时它就会激活了。 add-plugin :

```
vim9script

def HighlightedYank(hlgroup = 'IncSearch', duration = 300, in_visual = true)
  if v:event.operator ==? 'y'
    if !in_visual && visualmode() != null_string
      visualmode(1)
      return
    endif
    var [beg, end] = [getpos("'["), getpos("']")]
    var type = v:event.regtype ?? 'v'
    var pos = getregionpos(beg, end, {type: type})
    var end_offset = (type == 'V' || v:event.inclusive) ? 1 : 0
    var m = matchaddpos(hlgroup, pos->mapnew((_, v) => {
```

```
      var col_beg = v[0][2] + v[0][3]
      var col_end = v[1][2] + v[1][3] + end_offset
      return [v[0][1], col_beg, col_end - col_beg]
    }))
    var winid = win_getid()
    timer_start(duration, (_) => m->matchdelete(winid))
  endif
enddef

autocmd TextYankPost * HighlightedYank()
```

下一章: usr_90.txt 安装 Vim

VIM 用户手册 - by Bram Moolenaar
译者：lang2

安装 Vim

install

在你开始使用 Vim 之前你必须安装它。根据你的系统不同，安装可能很简单，也可能稍微复杂一点。这一章会给出一些提示，同时也叙述如何升级。

前一章： usr_52.txt 编写使用 Vim9 脚本的插件
目录： usr_toc.txt

90.1 Unix

首先你要决定的是：要为整个系统安装 Vim 还是为单个用户。安装过程几乎是一样的。但是 Vim 安装的路径不同。

对于系统安装来说基目录常使用 "/usr/local"。但对于你的系统可能也不同。看看其它的软件包是安装在哪里的。

当为单个用户安装时，你可以使用你的主目录作为基目录。Vim 的文件将被放置到 "bin" 和 "share/vim" 等子目录中。

从一个程序包安装

取决于使用的 Unix/Linux 系统，可能有预先编译好的可执行文件包。需要搜索一下。过去我们维护不同 UNIX 系统的列表，该列表已过期，所以已删除。

通过源代码来自己编译你自己的 UNIX 版本是个更好的办法。同时，从源代码创建 Vim 编辑器允许你控制可选的特性。但这需要一个编译器。

如果你有一个 Linux 的发行版本，其中的 "vi" 程序很可能是一个最小版的 Vim。例如，它可能不支持语法高亮。试试在你的发行版中找另外一个 Vim 程序包，或在网上搜搜。

从源码开始

你需要下面的东西来编译 Vim:

- 一个 C 编译器 (最好用 GCC 或 clang)
- git (可选，仅当从 github 克隆时才需要)
- zip/unzip (用来解压归档)
- Vim 源码归档文件

要下载 Vim 源代码，可以从 Github 项目页克隆：

```
git clone https://github.com/vim/vim.git
```

或直接下载归档：

```
https://github.com/vim/vim/archive/refs/heads/master.zip
```

编 译

先建立一个工作目录，例如：

```
mkdir ~/vim
cd ~/vim
```

在其中解开下载的文档。可以这样解压缩：

```
unzip vim-master.zip
```

如果你觉得缺省的特性就够了的话，照下面这样直接编译 Vim 就行了：

```
cd vim-master/src
make
```

make 程序会执行 configure 并编译所有的东西。后边我们会介绍如何将不同的特性编译进 Vim。

如果在编译时出现错误，请仔细的查看错误信息。编译程序会给出对于错误的提示。希望你能据此更正错误。你可能需要关闭一些特性。Makefile 也给出了一些可能适合你的特定系统的提示。

测 试

现在你可以试试你的编译成功了没有：

```
make test
```

这将执行一系列的测试脚本来确认 Vim 能正常的工作。测试的过程中 Vim 会多次启动，也会闪过各种各样的文本和信息。如果测试成功的话你最终会看到：

```
test results:
ALL DONE
```

如果你得到 "TEST FAILURE" 的信息，有些测试失败了。如果有一两个测试失败，Vim 可能还可以工作，不过不太完美。但是如果你看到大量的错误信息或者测试无法完成，那一定是有麻烦了。要么尝试自己解决，要么找个能帮助你的人。你可以在 maillist-archive 中查找解决办法。如果实在解决不了，你可以在 maillist 中提问看看有没有人能帮你。

安 装

install-home

如果你想安装在自己的 home 目录，编辑 Makefile 并查找这样的一行：

```
#prefix = $(HOME)
```

把行首的那个 # 去掉。
当你要为整个系统安装的时候，Vim 很可能已经为你选择好了一个合适的安装目录。你也可以照下面讲的自己选择一个，但你必须先改变身份为 root。

要安装 Vim 执行：

```
make install
```

这将把所有相关的文件转移到正确的地方。现在你可以试着运行一下 vim 以确认没有问题。下面两个简单的测试可以检查 Vim 能不能正确地找到那些运行时文件：

```
:help
:syntax enable
```

如果不成功的话，使用下面的命令来查看 Vim 是在哪儿找那些文件的：

```
:echo $VIMRUNTIME
```

你还可以用 "-V" 参数来启动 Vim，那样你可以得到更多的启动信息：

```
vim -V
```

别忘了本手册假设你用某一特定的方式使用 Vim。在安装完成之后，请依照 not-compatible 里面的指示来要求 Vim 以该方式运行。

选 择 特 性

Vim 有多种方法可以选择其特性。一个简单（直接）的办法就是编辑 Makefile。那里有很多的指令和例子。通常你只要去掉一行的注释就可以打开或关闭某一特性。
另外你也可以运行 "configure" 程序。这使得你可以手动指明你所希望的特性。缺点是你得知道在命令行上输入什么。
下面列出一些最可能引起你兴趣的 configure 参数。同样的，你也可以通过编辑 Makefile 来设定。

--prefix={directory}	安装 Vim 的顶级目录。
--with-features=tiny	关掉一些特性。
--with-features=normal	打开更多的特性。
--with-features=huge	打开大多数特性。 参阅 +feature-list 察看每一种方式的详细特性列表。
--enable-perlinterp	开启 Perl 接口。类似的还有为 ruby、python 和 tcl 准备的参数。
--disable-gui	不编译 GUI 界面。
--without-x	不编译 X-windows 相关的特性。 当两个同时使用的时候，Vim 不会连接 X 服务器，这会使启动更快些。

察看所有参数的列表使用：

```
./configure --help
```

这里你可以找到各个特性的一些解释，以及指向其它更多信息的链接： feature-list 。
对于那些喜欢冒险的人，你可以编辑 "feature.h" 文件。你还可以自己改动源代码！

90.2 MS-Windows

有两种方法可以安装 Vim 的 Microsoft Windows 版本。你可以选择自己解压缩几个文档，或者使用一个自动安装程序。多数使用较新的电脑的用户会选择第二种方法。对于第一种方法你需要：

- 包含 Vim 可执行程序的文档。
- Vim 运行时归档文件。
- 解压缩程序。

下面这个文件提供了一个镜像列表，它可以帮助你找到距离你最近的镜像从而提高下载的速度：

```
ftp://ftp.vim.org/pub/vim/MIRRORS
```

如果你觉得够快的话，或者就用主站 ftp.vim.org。进入到 "pc" 目录就可以找到一组文件。文件的名字中含有版本名。最好下载最新的版本。这里我们用 "82" 为例，表示 8.2 版。

```
gvim82.exe              自动安装程序。
```

第二种方法你就只需要这个了。执行该程序并依照提示操作就可以了。

第一种方法你需要选择下列的其中一个可执行文档：

```
gvim82.zip              一般的 MS-Windows GUI 版本。
gvim82ole.zip           带有 OLE 支持的 MS-Windows GUI 版本。
                        更消耗内存，支持与其他 OLE 程序的接口。
vim82w32.zip            32 位 MS-Windows 控制台版本。
```

你只需要其中的一个就够了。不过你可以同时安装一个 GUI 版本和一个控制台版本。但是包含有运行时文件的归档文件总是必需的。

```
vim82rt.zip             运行时文件。
```

使用你的解压缩程序把文件解出来。例如，使用 "unzip" 程序：

```
cd c:\
unzip path\gvim82.zip
unzip path\vim82rt.zip
```

这将把文件解压缩到 "c:\vim\vim82" 目录。如果你已经有一个 "vim" 目录了，你需要进入其上级目录。
现在进入那个 "vim\vim82" 目录，然后执行安装程序：

```
install
```

仔细的察看并选择你想要的。当你最终选择 "do it" 的时候安装程序将根据你的选项执行安装的步骤。
安装程序不会移走运行时文件。它们会被原封不动留在你解压缩的地方。

如果你对可执行文件包含的特性不满意的话，你可以试着自己编译 Vim。从你取得可执行文档同样的地方取得源代码。你需要一个有对应的 makefile 文件的编译器。可以使用 Microsoft Visual C、MingW 和 Cygwin 编译器。相关提示要查阅 src/INSTALLpc.txt。

90.3 升级

如果你已经有一个版本的 Vim 并想安装另一个，这里就是你需要做的。

UNIX

当你键入 "make install" 命令时，运行时文件将被拷贝到一个该版本专有的目录，而不会覆盖现存版本的文件。这样就便可以同时运行两个版本的 Vim。
但是可执行文件 "vim" 将覆盖老的版本。如果你不在乎保留老的版本，这样就没问题。你还可以手动删除那些老版本的运行时文件。只要删除掉那个版本所在目录及其下所有文件即可。例:

```
rm -rf /usr/local/share/vim/vim74
```

通常该目录里不会有你改动过的文件。如果你的确改动了，例如，"filetype.vim" 文件，你最好把改动合并到新的版本然后在删除老的文件。

如果你比较小心，希望先试试新的版本，你可以用另外的名字安装新的版本。你需要指定一个 configure 参数。例如:

```
./configure --with-vim-name=vim8
```

在运行 "make install" 之前，你可以使用 "make -n install" 命令来查看安装会不会覆盖现有的重要文件。
当你最终决定启用新版本的时候，你要做的仅仅是将可执行文件改名为 "vim"。例:

```
mv /usr/local/bin/vim8 /usr/local/bin/vim
```

MS-WINDOWS

升级几乎和安装一个新版本一样。把新版本的文件解压缩到上一版本的同样位置。会对于新版本的文件生成一个新的目录。例如: "vim82"。你的运行时文件，vimrc 文件，viminfo 文件等等，会被原样保留。
如果你想同时运行两个版本，你就得做点作些手工活了。别运行安装程序。那样会覆盖老版本的一些文件。用全路径来运行新的可执行文件。程序会自动找到它需要的运行时文件。然而，如果你在别的地方设定了 $VIMRUNTIME 的值，这个法子就不灵了。
如果你对升级满意的话，你可以删除掉前一版本的文件。参阅 90.5 。

90.4 常见安装问题

这一节列举一些安装过程中常见的麻烦并给出一些解决办法。同时也回答一些安装相关的问题。

Q: 我没有 Root 权限。怎么安装 Vim? (Unix)

使用下面的配置命令可以将 Vim 安装到目录 $HOME/vim:

```
./configure --prefix=$HOME
```

这样你可以安装一个个人拷贝。请将 $HOME/bin 加入 path 中，以便执行本编辑器。参阅 install-home 。

Q: 我屏幕上的颜色不对。(Unix)

用下面的命令检查终端的设置:

```
echo $TERM
```

如果列出的终端类型不正确，更正之。查阅 06.2 可以得到更多的提示。另一个解决办法是使用 Vim 的 GUI 版本。名为 gvim。这样你就不需要一个正确的终端设置了。

Q: 我的 Backspace 和 Delete 键工作不正常

对于这两个键 (<BS>、<Del>)，什么键产生它们的键码不是很清楚。首先检查一下你的 $TERM 设置。如果没什么错误，试试这个:

```
:set t_kb=^V<BS>
:set t_kD=^V<Del>
```

第一行你需要键入 CTRL-V 在按 backspace 键。第二行你需要键入 CTRL-V 再按 Delete 键。你可以把这两行加入到你的 vimrc 文件中，参见 05.1 。 这样做的一个缺点是如果你换用另外一个终端程序的话，这些设置可能就不起作用了。其他的解决办法可以在 :fixdel 中找到。

Q: 我使用 RedHat Linux。我能使用系统自带的 Vim 吗?

缺省情况下 RedHat 仅安装一个 Vim 的最小版本。找一个叫 "Vim-enhanced-version.rpm" 的安装包就行了。

Q: 我怎么才能打开语法高亮? 怎么才能使用插件?

使用那个 vimrc 脚本的例子。在这儿你可以找到详细地解释: not-compatible 。

在第 6 章阅读有关语法高亮的解释: usr_06.txt 。

Q: 有没有一个好的 vimrc 文件可以使用?

到 www.vim.org 网站去找找。

Q: 哪儿可以找到好的 Vim 插件?

看看 Vim-online 网站: 。很多用户在那里上载了非常有用的 Vim 脚本。

Q: 哪有更多的技巧提示?

看看 Vim-online 网站: 。那里收集了很多用户的提示。你也可以在 maillist-archive 里查找。

90.5 卸载 Vim

虽然这几乎不可能发生，但是万一你希望完全卸载 Vim，本节告诉你怎么做。

UNIX

如果你是通过一个软件包安装 Vim 的，看看你的软件包管理器是如何卸载一个软件包的。如果你是通过源代码安装的你可以用这个命令卸载:

```
make uninstall
```

然而，如果你已经将那些文件删除或者你使用的是别人提供的归档，你就没法这样做了。那就只能手动删除那些文件。这里给出一个例子，假设你是以 root 身份将 Vim 安装到 "/usr/local" 下的:

```
rm -rf /usr/local/share/vim/vim82
rm /usr/local/bin/eview
rm /usr/local/bin/evim
rm /usr/local/bin/ex
rm /usr/local/bin/gview
rm /usr/local/bin/gvim
rm /usr/local/bin/gvim
rm /usr/local/bin/gvimdiff
rm /usr/local/bin/rgview
rm /usr/local/bin/rgvim
rm /usr/local/bin/rview
rm /usr/local/bin/rvim
rm /usr/local/bin/rvim
rm /usr/local/bin/view
rm /usr/local/bin/vim
rm /usr/local/bin/vimdiff
rm /usr/local/bin/vimtutor
rm /usr/local/bin/xxd
rm /usr/local/man/man1/eview.1
rm /usr/local/man/man1/evim.1
rm /usr/local/man/man1/ex.1
rm /usr/local/man/man1/gview.1
rm /usr/local/man/man1/gvim.1
rm /usr/local/man/man1/gvimdiff.1
```

```
rm /usr/local/man/man1/rgview.1
rm /usr/local/man/man1/rgvim.1
rm /usr/local/man/man1/rview.1
rm /usr/local/man/man1/rvim.1
rm /usr/local/man/man1/view.1
rm /usr/local/man/man1/vim.1
rm /usr/local/man/man1/vimdiff.1
rm /usr/local/man/man1/vimtutor.1
rm /usr/local/man/man1/xxd.1
```

MS-WINDOWS

如果你是用那个自动安装程序安装的话你可以运行 Vim 目录下 (例如 "c:\vim\vim82") 的 "uninstall-gui" 程序。你也可以通过开始菜单来运行。这可以删除大多数的文件菜单以及桌面快捷方式。有些文件可能只有待 Windows 重新启动之后才能被删除。
你可能被提示来删除整个 "vim" 目录。那里面很可能有你的 vimrc 文件以及其他你创建的运行时文件。要当心。

否则，如果你使用了那些 zip 归档安装了 Vim 的话，最好使用 "uninstall" 程序。它就在 "install" 程序所在的目录， 例如，"c:\vim\vim82"。用通常的那个 "install/remove software" 页也一样有效。
然而，这只会清除 Vim 的注册表信息。你得自己删除硬盘上的那些文件。简单的选中 "vim\vim82" 目录并删除就可以了。那里边不应该包含任何你所改动过的文件，不过你可能希望先检查一下。
"vim" 目录很有可能包含有你的 vimrc 文件以及其它你创建的运行时文件。你可能想保留它。

目 录: usr_toc.txt

 vim:tw=78:ts=8:noet:ft=help:norl: