motion.txt For Vim version 7.4. 最近修改: 2013 年 8 月

VIM 参考手册 by Bram Moolenaar

译者: Willis

http://vimcdoc.sf.net

光标移动 *cursor-motions* *navigation*

这些命令用于移动光标。如果新的位置离开了当前显示范围，屏幕将滚动到合适的位置，使得光标可见（参见 'scrolljump' 和 'scrolloff' 选项)。

概论:

如果你想知道在当前文件所在的位置，可以用 "CTRL-G" 命令 |CTRL-G| 或者 "g CTRL-G" 命令 |g_CTRL-G|。如果你置位了 'ruler' 选项，光标位置会在状态行上保

持更新（Vim 会因此稍慢一些)。

有经验的用户更喜欢 hjkl 键，因为它们就在自己的手指下面。初学者则倾向于光标移动键，因为他们不知道 hjkl 键做什么。看看键盘的布局就很容易记住 hjkl 的意义。把 j 想象成一个向下的箭头就可以了。

置位 'virtualedit' 选项使得光标可以移动到还没有字符或者半个字符的位置。

==========

1. 动作和操作符 *operator*

动作命令出现在操作符之后，从而使操作符作用于被该动作所跨越的文本之上。也就是，在动作之前和之后的光标位置之间的文本。一般的，操作符用来删除或者改变文本。下面列出所有的操作符:

|c| c 修改（change)

|d| d 删除（delete)

|y| y 抽出（yank）到寄存器（不改变文本)

|~| ~ 变换大小写（只有当 'tildeop' 置位时有效)

|g~| g~ 变换大小写

|gu|gu 变为小写

|gU| gU 变为大写

|!| ! 通过外部程序过滤

|=| = 通过 'equalprg' (若为空，C-indenting）过滤

|gq|gq 文本排版

|g?| g? ROT13 编码

|>| > 右移

|<| < 左移

|zf| zf 定义折叠

|g@| g@ 调用 'operatorfunc' 选项定义的函数

如果动作包括一个次数而操作符之前也有一个的话，两者相乘。因此，"2d3w" 删除六个单词。

大多数情况下，光标在应用操作符后停在被操作的文本的起始处。例如，"yfe" 不移动光标，而 "yFe" 则向左移动光标到抽出的文本的起始的那个 "e" 上。

linewise *characterwise*

操作符或者影响开始和结束位置之间的整行或者字符区间。一般说来，在行间移动的动作影响整行（或者说面向行的），而在行内移动的动作影响字符区间（或者说面向字符的）。但是有例外。

exclusive *inclusive*

面向字符的动作或者是闭的，或者是开的。闭动作的开始和结束位置包含在操作范围里。开动作中，靠近缓冲区尾端的最后一个字符不被包含在内。行的动作总包含开始和结尾的位置。

下面的命令里总会提到动作是行的、开的、或是闭的。不过，有两个一般的特例:

1. 如果一个开动作的结尾应在第一列，那么，它会前移到上一行的结尾处，并成为一个闭动作。例如，"}" 移动到一个段落之后的第一行。但 "d}" 不会包含那一行。

exclusive-linewise

2. 如果一个开动作的结尾应在第一列，而它的开始在一行的第一个非空白字符之前或之上，那么它会转换成一个行动作。例如，如果一个段落以若干空白开始，而你在第一个非空白字符上执行 "d}"，那么该段落的所有行都被删除，包括之前的空白。如果你再放置（put)，那么删除的行将插在光标之下。

请 注意 如果操作符处于等待状态（键入了操作符命令但还没有键入动作)，将应用一组特别的映射命令。参见 |:omap|。

除了先给出操作符再给出动作的方式以外，你还可以用可视模式: 先用 "v" 标记文本的开始处，移动光标到文本的末尾，然后再输入操作符。开始和当前光标位置之间的文本以高亮显示，从而你可以直观地看到要操作的文本。这种方式提供了更多的自由，但代价是更多的键击而且重做不易。参见可视模式的章节 |Visual-mode|。

你可以用 ":" 命令定义一个动作。例如，"d:call FindEnd()"。

不过如果命令多于一行，该操作不能用 "." 重复。例如，可以重复: >

```
d:call search("f")<CR>
```

但不能重复: >

```
d:if 1<CR>
    call search("f")<CR>
endif<CR>
```

注意 ":" 的使用使任何动作变成面向字符。

强 制 一 个 动 作 面 向 行、面 向 字 符 或 者 面 向 可 视 列 块

如果一个动作不是你希望使用的类型，你可以在操作符后用 "v"、"V" 或者 CTRL-V 来强制转换类型。

示例: >

dj

删除两行 >

dvj

删除当前光标位置到光标下方字符之间字符 >

d<C-V>j

删除光标所在和光标下方的字符。

把面向行的动作强制为面向字符或者列块时要注意，列不一定总有定义。

o_v

v 在操作符后和动作命令之前应用: 即使该动作是面向行的，也强制该操作面向字符。如果它应是面向行的，则变成开动作。如果已经是面向字符的，则在开/闭间切换。这可以使一个开动作成为闭的，或者使一个闭动作成为开的。

o_V

V 在操作符后和动作命令之前应用: 即使该动作是面向字符的，也强制该操作面向行。

o_CTRL-V

CTRL-V 在操作符后和动作命令之前应用: 强制该动作面向列块。这类似于可视列块模式的选择，由动作之前和之后的光标位置定义两个角落。

==========

2. 左右动作 *left-right-motions*

这些命令移动光标到当前行的指定列。除了 "$" 之外（有可能跨越多行），它们最多停在

该行的第一列或者行尾。'whichwrap' 选项可以使其中的一些命令跨越行的边界。

h 或 *h*

<Left> 或 *<Left>*

CTRL-H 或 *CTRL-H* *<BS>*

<BS> 向左 [count] 个字符。|exclusive| 开动作。

注意: 如果你希望 <BS> 删除字符，使用如下映射:

:map CTRL-V<BS> X

(要输入 "CTRL-V<BS>" 先敲 CTRL-V 键，紧跟一个 <BS> 键)

如果 <BS> 和希望的操作不符，查阅 |:fixdel|。

l 或 *l*

<Right> 或 *<Right>* *<Space>*

<Space> 向右 [count] 个字符。|exclusive| 开动作。

0

0 到行的第一个字符。|exclusive| 开动作。

<Home> *<kHome>*

<Home> 到行的第一个字符。|exclusive| 开动作。上下移动时，停在

相同的文本列 (如果可能的话)，与之对照，多数其它命令则

保持在相同的_屏幕_列上。和 "1|" 类似。如果一行以 <Tab>

开头，则和 "0" 有区别。{Vi 无此功能}。

^

^ 到行的第一个非空白字符。|exclusive| 开动作。

$ *<End>* *<kEnd>*

$ 或 <End>　　到行尾。如果给出 [count]，则先往下走 [count-1] 行。

|inclusive| 闭动作。可视模式下，光标移到紧贴该行最后一

个字符之后的位置。

如果置位了 'virtualedit'，"$" 可从行尾之后的空白后退到

行尾。

*g_*

g_　　往下 [count-1] 行并到该行的最后一个非空白字符。

|inclusive| 闭动作。{Vi 无此功能}

g0 *g<Home>*

g0 或 g<Home>　　如果设置了行回绕 ('wrap' 打开): 到屏幕行的第一个字符。

|exclusive| 开动作。如果一文本行不能在屏幕上完全显示，

它和 "0" 就会有不同。

如果没有设置 ('wrap' 关闭): 到该行屏幕显示范围的最左

面的字符。如果该行第一个字符不在屏幕上，则它和 "0" 就

有不同。{Vi 无此功能}

*g^*

g^　　如果设置了行回绕 ('wrap' 打开): 到屏幕行的第一个非空白

字符。|exclusive| 开动作。如果一文本行不能在屏幕上完全

显示，它和 "^" 就会有区别。

如果没有设置 ('wrap' 关闭): 到该行屏幕显示范围的最左面

的非空白字符。如果该行第一个字符不在屏幕上，则它和 "^"

就有区别。{Vi 无此功能}

gm

gm 和 "g0" 类似，但（尽可能）向右移到屏幕显示宽度的中间位置。{Vi 无此功能}

g$ *g<End>*

g$ 或 g<End> 如果设置了行回绕（'wrap' 打开）：往下 [count-1] 屏幕行并到该屏幕行的行尾。|inclusive| 闭动作。如果一文本行不能在屏幕上完全显示，它和 "$" 就会有不同。
如果没有设置（'wrap' 关闭）：到该行屏幕显示范围的最右面的字符。如果该行的行尾字符不可见，则它和 "$" 就有不同。另外，垂直移动保持所在的列，而不是移到行尾。
打开 'virtualedit' 时移动到屏幕行尾。
{Vi 无此功能}

bar

| 到当前行的 [count] 屏幕列。|exclusive| 开动作。
Ceci n'est pas une pipe (译者注：法语 这不是一只烟斗，这是名句，pipe 也做管道讲，有双关义)。

f

f{char} 到右侧第 [count] 次出现的字符 {char}。光标放在 {char} 上。|inclusive| 闭动作。
{char} 可以输入二合字母 |digraph-arg|。
如果 'encoding' 设为 Unicode，则可以输入合成用字符，参见 |utf-8-char-arg|。

{char} 要进行 |:lmap| 映射。插入模式下的 CTRL-^ 命令可
以用来切换这一点 |i_CTRL-^|。

F

F{char} 到左侧第 [count] 次出现的字符 {char}。
光标放在 {char} 上。|exclusive| 开动作。
{char} 可以和 |f| 命令相同的方式输入。

t

t{char} 直到右侧第 [count] 次出现的字符 {char} 之前。光标放在
{char} 左边的位置。|inclusive| 闭动作。
{char} 可以和 |f| 命令相同的方式输入。

T

T{char} 直到左侧第 [count] 次出现的字符 {char} 之后。光标放在
{char} 右侧的位置。|exclusive| 开动作。
{char} 可以和 |f| 命令相同的方式输入。

;

; 重复上次的 f、t、F 或者 T 命令 [count] 次。见 |cpo-;|

,

, 反方向重复上次的 f、t、F 或者 T 命令 [count] 次。另见
|cpo-;|

==========

3. 上下动作 *up-down-motions*

k 或 *k*

<Up>　　或　　*<Up>* *CTRL-P*

CTRL-P　　向上 [count] 行。|linewise| 行动作。

j　　或　　*j*

<Down>　　或　　*<Down>*

CTRL-J　　或　　*CTRL-J*

<NL>　　或　　*<NL>* *CTRL-N*

CTRL-N　　向下 [count] 行。|linewise| 行动作。

gk　　或　　*gk* *g<Up>*

g<Up>　　向上 [count] 显示行。|exclusive| 开动作。

在行回绕和操作符之后与 'k' 不同，因为后者不是面向行

的。{Vi 无此功能}

gj　　或　　*gj* *g<Down>*

g<Down>　　向下 [count] 显示行。|exclusive| 开动作。

在行回绕和操作符之后与 'j' 不同，因为后者不是面向行

的。{Vi 无此功能}

-

- <minus>　　向上 [count] 行，停在第一个非空白字符上。|linewise|

行动作。

+　　或　　*+*

CTRL-M　　或　　*CTRL-M* *<CR>*

<CR>　　向下 [count] 行，停在第一个非空白字符上。|linewise| 行动作。

_

_　<underscore>　　向下 [count] - 1 行，停在第一个非空白字符上。|linewise| 行动作。

G

G　　到第 [count] 行，缺省是最后一行， |linewise| 行动作。
如果 'startofline' 没有置位，保持在相同的列上，不然，停在第一个非空白字符上。
G 是 |jump-motions| 之一。

<C-End>

<C-End>　　到第 [count] 行，缺省是最后一行；并停在最后一个字符上。|inclusive| 闭动作。{Vi 无此功能}

<C-Home> 或　　*gg* *<C-Home>*

gg　　到第 [count] 行，缺省是第一行， |linewise| 行动作。
如果 'startofline' 没有置位，保持在相同的列上，不然，停在第一个非空白字符上。

:[range]

:[range]　　把光标移到 [range] 的最后一行。[range] 也可以是单独一个行号，如 ":1" 或 ":'m"。
和 |G| 不同，此命令不修改 |jumplist|。

N%

{count}%　　到文件的 {count} 百分比处，|linewise| 行动作。

新的行号计算方法如下:

({count} * 总行数 + 99) / 100

如果 'startofline' 没有置位，保持在相同的列上，不然，

停在第一个非空白字符上。{Vi 无此功能}

:[range]go[to] [count]　　*:go* *:goto* *go*

[count]go　　到缓冲区的第 {count} 个字节。缺省的 [count] 是 1，即文

件开始处。如果给定 [range]，则最后的一个数字用作字节的

序号。'fileformat' 的当前设置决定如何计算换行符的个

数。

另见 |line2byte()| 函数和 'statusline' 的 'o' 选项。

{Vi 无此功能}

{仅当编译时加入 |+byte_offset| 特性才有效}

这些命令移动到特定的行上，最多，它们移动到第一行或者最后一行。开始两个命令的光标将停留在最后一个改变列号命令所指定的列上 (如果可能的话)。"$" 命令除外，这时，光标总会停在一行的行尾。

如果 "k"、"-" 或 CTRL-P 使用时给出 [count]，而光标上方的行数小于 [count] 且 'cpo' 选项包含了 "-" 标志位，报错 |cpo--|。

==========

4. 单词动作　　*word-motions*

<S-Right> 或 *<S-Right>* *w*

w 正向 [count] 个单词。|exclusive| 开动作。

<C-Right> 或 *<C-Right>* *W*

W 正向 [count] 个字串。|exclusive| 开动作。

e

e 正向到第 [count] 个单词的尾部。|inclusive| 闭动作。

不停留在空行上。

E

E 正向到第 [count] 个字串的尾部。|inclusive| 闭动作。

不停留在空行上。

<S-Left> 或 *<S-Left>* *b*

b 反向 [count] 个单词。|exclusive| 开动作。

<C-Left> 或 *<C-Left>* *B*

B 反向 [count] 个字串。|exclusive| 开动作。

ge

ge 反向到第 [count] 个单词的尾部。|inclusive| 闭动作。

gE

gE 反向到第 [count] 个字串的尾部。|inclusive| 闭动作。

这些命令在单词或字串间移动。

word

一个单词由字符、数字和下划线序列或者其他的非空白字符的序列组成。单词间可以空白

字符（空格、制表、换行）分隔。这一规则可以用 'iskeyword' 选项改变。空行也被认作单词。

WORD

一个字串由非空白字符序列组成。字串以空白分隔。空行也被认作字串。

已折叠的行序列被认作由单个字符组成的单词。"w" 和 "W"、"e" 和 "E" 移动到折叠行之后的第一个单词或字串的开始/结尾处。"b" 和 "B" 移动到折叠之前的第一个单词的开始处。

特例: 如果光标在非空白字符上，"cw" 和 "cW" 等价于 "ce" 和 "cE"。这是因为 "cw" 被诠释为 修改-单词 (change-word)，而单词并不包括后续的空格。{Vi: 在多个空白的某一个空白之上做 "cw" 只会改动单个空白字符；这可能不太对，因为 "dw" 是删除所有后面的空白的}

另外一个特例: 如果 "w" 动作带操作符并且该动作的最后一个单词在行尾，则该操作范围结束于行尾而非下一行的第一个单词。

原始 Vi 的 "e" 实现有问题。例如，如果前一行为空而光标停在后一行的第一个字符的话，"e" 就会卡在那里，但是你用 "2e" 就很正常。Vim 里就不是，"ee" 和 "2e" 有相同的行为，这更合乎逻辑。不过，这造成了 Vi 和 Vim 小小的不兼容。

==========

5. 文本对象动作 *object-motions*

(

(反向 [count] 个句子。|exclusive| 开动作。

)

)　　正向 [count] 个句子。|exclusive| 开动作。

{

{　　反向 [count] 个段落。|exclusive| 开动作。

}

}　　正向 [count] 个段落。|exclusive| 开动作。

]]

]]　　正向 [count] 个小节或到出现在首列的 '{'。如果带操作符，则同时停留在首列的 '}' 的下方。|exclusive| 开动作。注意 |exclusive-linewise| 常常适用。

][

][　　正向 [count] 个小节或到出现在首列的 '}'。|exclusive| 开动作。

注意 |exclusive-linewise| 常常适用。

[[

[[　　反向 [count] 个小节或到出现在首列的 '{'。|exclusive| 开动作。

注意 |exclusive-linewise| 常常适用。

[]

[]　　反向 [count] 个小节或到出现在首列的 '}'。|exclusive| 开动作。

注意 |exclusive-linewise| 常常适用。

这些命令在三类文本对象上移动，见下。

sentence

一个句子以 '.'、'!' 或者 '?' 结尾并紧随着一个换行符、空格或者制表符。标点和空白字符之间可以出现任何数量的闭括号和引号: ')'、']'、'"' 和 '''。另，段落和小节的边界也视为句子的边界。

如果 'cpoptions' 包含 'J' 标志位，那么标点之后的空格至少要出现两个，而且制表符不被视为空白字符。

paragraph

一个段落从空行或某一个段落宏命令开始，段落宏由 'paragraphs' 选项里成对出现的字符所定义。它的缺省值为 "IPLPPPQPP TPHPLIPpLpItpplpipbp"，也就是宏 ".IP"、".LP" 等 (这些是 nroff 宏，所以句号一定要出现在第一列)。小节边界也被视为段落边界。

注意 空白行 (只包含空白) _不是_ 段落边界。

也要 注意: 这不包括首列出现的 '{' 或 '}'。如果 'cpoptions' 里包含 '{' 标志位，那么首列的 '{' 用作段落边界 |posix|。

section

一个小节从首列出现的换页符 (<C-L>) 或某一个小节宏命令开始。小节宏由 'sections' 选项里成对出现的字符所定义。它的缺省值是 "SHNHH HUnhsh"，也就是说小节可以从如

下的 nroff 宏开始: ".SH"、".NH"、".H"、".HU"、".nh" 和 ".sh"。

"]" 和 "[" 命令也停在首列出现的 '{' 或 '}' 上。这有助于在 C 程序里找到函数的开始和结束位置。 注意: 命令的第一个字符决定搜索的方向，第二个字符决定要找到的括号。

如果你的 '{' 或 '}' 不在第一列但是你还是希望用 "[[" 和 "]]" 来找它们，试试这些

映射: >

```
:map [[ ?{<CR>w99[{
:map ][ /}<CR>b99]}
:map ]] j0[[%/{<CR>
:map [] k$][%?}<CR>
```

[照文本直接输入，参见 |<>|]

==========

6. 文本对象选择 *object-select* *text-objects*

v_a *v_i*

这里是一系列只能在可视模式或操作符后使用的命令。这些命令或以 "a" 打头，代表选择一个（"a"n）包含空白的对象；或以 "i" 带头，代表选择内含（"i"nner）对象: 它们不包含空白。另外，空白本身也是内含对象。这样，"内含" 对象总比 "一个" 对象选择较少的文本。

这些命令都是 {Vi 无此功能}。

这些命令只有在编译时加入 |+textobjects| 特性后才有效。

另见 `gn` 和 `gN`，操作对象是前次搜索模式。

v_aw *aw*

aw "一个单词"，选择 [count] 个单词（见 |word|）。

包括开头或拖尾的空白，但不单独计算。在可视面向行的模式下，"aw" 切换到可视面向字符的模式。

v_iw *iw*

iw "内含单词"，选择 [count] 个单词（见 |word|）。

单词之间的空白也被算为一个单词。在可视面向行的模式下，

"iw" 切换到可视面向字符的模式。

v_aW *aW*

aW "一个字串"，选择 [count] 个字串（见 |WORD|）。

包括开头或拖尾的空白，但不单独计算。在可视面向行的模式

下，"aW" 切换到可视面向字符的模式。

v_iW *iW*

iW "内含字串"，选择 [count] 个字串（见 |WORD|）。

字串之间的空白也被算为一个字串。在可视面向行的模式下，

"iW" 切换到可视面向字符的模式。

v_as *as*

as "一个句子"，选择 [count] 个句子（见 |sentence|）。

可视模式下它切换为面向字符的模式。

v_is *is*

is "内含句子"，选择 [count] 个句子（见 |sentence|）。

可视模式下它切换为面向字符的模式。

v_ap *ap*

ap "一个段落"，选择 [count] 个段落（见 |paragraph|）。

特例: 空白行（只包含空白的行）也被视为段落边界。

可视模式下它切换为面向行的模式。

v_ip *ip*

ip "内含段落"，选择 [count] 个段落（见 |paragraph|）。

特例: 空白行（只包含空白的行）也被视为段落边界。

可视模式下它切换为面向行的模式。

a] *v_a]* *v_a[* *a]* *a[*

a["一个 [] 块"，选择 [count] 层 '[' ']' 块。为此，先反向

查找第 [count] 个未匹配的 '['，然后查找其相应的 ']'。

两者之间的文本，包括 '[' 和 ']'，都被选择。

可视模式下它切换为面向字符的模式。

i] *v_i]* *v_i[* *i]* *i[*

i["内含 [] 块"，选择 [count] 层 '[' ']' 块。为此，先反向

查找第 [count] 个未匹配的 '['，然后查找其相应的 ']'。

两者之间的文本，但不包括 '[' 和 ']'，被选择。

可视模式下它切换为面向字符的模式。

a) *v_a)* *a)* *a(*
a(*v_ab* *v_a(* *ab*

ab "一个块"，选择 [count] 层块，从 "[count] [(" 到其相应

的 ')'，包括 '(' 和 ')' (见 |[(|)。它并不包括括号之

外的空白。

可视模式下它切换为面向字符的模式。

i) *v_i)* *i)* *i(*
i(*v_ib* *v_i(* *ib*

ib "内含块"，选择 [count] 层块，从 "[count] [(" 到其相应

的 ')'，但不包括 '(' 和 ')' (见 |[(|)。

可视模式下它切换为面向字符的模式。

a>　　*v_a>* *v_a<* *a>* *a<*

a<　　"一个 <> 块"，选择 [count] 层 <> 块，从反向第 [count] 个未匹配的 '<' 到其匹配的 '>'，包括 '<' 和 '>'。可视模式下它切换为面向字符的模式。

i>　　*v_i>* *v_i<* *i>* *i<*

i<　　"内含 <> 块"，选择 [count] 层 <> 块，从反向第 [count] 个未匹配的 '<' 到其匹配的 '>'，但不包括 '<' 和 '>'。可视模式下它切换为面向字符的模式。

v_at *at*

at　　"一个标签块"，选择 [count] 层标签块。从反向第 [count] 个未匹配的 "<aaa>" 到其匹配的 "</aaa>"，包括 "<aaa>" 和 "</aaa>"。详情见 |tag-blocks|。可视模式下它切换为面向字符的模式。

v_it *it*

it　　"内部标签块"，选择 [count] 层标签块。从反向第 [count] 个未匹配的 "<aaa>" 到其匹配的 "</aaa>"，但不包括 "<aaa>" 和 "</aaa>"。详情见 |tag-blocks|。可视模式下它切换为面向字符的模式。

a}　　*v_a}* *a}* *a{*
a{　　*v_aB* *v_a{* *aB*

aB　　"一个大块"，选择 [count] 层大块，从 "[count] [{" 到其

相应的 '}'，包括 '{' 和 '}' (见 |[{|)。

可视模式下它切换为面向字符的模式。

i}　　*v_i}* *i}* *i{*
i{　　*v_iB* *v_i{* *iB*

iB　　"内含大块"，选择 [count] 层大块，从 "[count] [{" 到其

相应的 '}'，但不包括 '{' 和 '}' (见 |[{|)。

可视模式下它切换为面向字符的模式。

a"　　*v_aquote* *aquote*
a'　　*v_a'* *a'*
a`　　*v_a`* *a`*

"一个引号字符串"。选择上一个引号开始到下一个引号结束的

文本。'quoteescape' 选项用于跳过转义的引号。

只在同一行内有效。

如果开始时光标在引号上，Vim 会从该行行首开始搜索，以决

定哪个引号对构成字符串。

包含任何拖尾的空白，如果没有拖尾的，也包含开头的空白。

可视模式下它切换为面向字符的模式。

在可视模式下重复此对象会包含另一个字符串。目前不使用计

数。

i"　　*v_iquote* *iquote*
i'　　*v_i'* *i'*
i`　　*v_i`* *i`*

类似于 a"、a' 和 a`，但不包括引号，而重复也不会扩展可

视选择区。

特例: 计数为 2 时包含引号，但不包括 a"/a'/a` 包含的额

外的空白。

在操作符之后:

非块对象:

对于 "一个" 命令: 操作符作用于对象与其后的空白。如果其后没有空白或者光标位于对象之前的空白上的话，那么也包括对象之前的空白。

对于 "内含" 命令: 如果光标在对象之上，那么操作符作用于该对象。如果光标在空白上，那么操作符作用于空白。

块对象:

操作符作用于光标所在位置所在（包括光标在括号上的特殊情况）的块。对于 "内含" 命令，不包含包围的括号。而对于 "一个" 命令，则包含之。

在可视模式下:

如果可视区域的起始和结束点在相同位置（刚输入 "v"):

选择一个区域，就和使用操作符一样。

如果可视区域的起始和结束点不同:

若不是块对象，该区域被对象或者下一个对象之前空白所扩展，对于 "一个" 命令，则两者都包含。扩展的方向决定于可视区域和光标的相对位置。

若是块对象，该块向外扩展一层。

让我们用一系列删除命令来说明，删除的范围从小到大。请 注意 对于单个字符和整行的操作我们用了已有的 Vi 移动命令。

"dl" 删除字符（缩写: "x"） |dl|

"diw" 删除内含单词 *diw*

"daw" 删除一个单词 *daw*

"diW" 删除内含字串（见 |WORD|） *diW*

"daW" 删除一个字串（见 |WORD|） *daW*

"dgn" 删除下一个前次搜索模式的匹配 *dgn*

"dd" 删除一行 |dd|

"dis" 删除内含句子 *dis*

"das" 删除一个句子 *das*

"dib" 删除内含 '(' ')' 块 *dib*

"dab" 删除一个 '(' ')' 块 *dab*

"dip" 删除内含段落 *dip*

"dap" 删除一个段落 *dap*

"diB" 删除内含 '{' '}' 大块 *diB*

"daB" 删除一个 '{' '}' 大块 *daB*

请 注意 移动命令和文本对象的区别。移动命令作用于这里（光标当前位置）到移动后的位置。而对象的使用则会作用于整个对象，而和光标在对象的何处无关。例如，我们可以比较 "dw" 和 "daw": "dw" 删除光标位置到下一个单词的起始处，"daw" 删除光标所在的整个单词和其后或其前的空白。

标签块 *tag-blocks*

"it" 和 "at" 文本对象尝试选择 HTML 和 XML 的匹配标签之间的块。但因为它们并不完全兼容，有一些限制。

通常的方法是选择 <tag> 直到匹配的 </tag> 为止的内容。"at" 包含标签，"it" 不包含。不过重复 "it" 时标签仍然会包含 (不然不会有任何改变)。此外，在没有内容的标签块上使用 "it" 会单独选择引导标签。

跳过 "<aaa/>" 项目。忽略大小写，即使对大小写应该敏感的 XML 也是如此。

HTML 里可以有
 或 <meta ...> 这样没有匹配结束标签的标签。它们被忽略。

这些文本对象能够容忍错误。单独出现的结束标签被忽略。

==

7. 位置标记 *mark-motions* *E20* *E78*

跳转到一个位置标记有两种方法:

1. 用 ` (反引号): 光标放在指定的位置，|exclusive| 开动作。

2. 用 ' (单引号): 光标放在指定位置所在行的第一个非空字符上，|linewise| 行动作。

m *mark* *Mark*

m{a-zA-Z} 把位置标记 {a-zA-Z} 设在当前光标位置 (不移动光标，这不是动作命令)。

m' *m`*

m' 或 m` 设置前次上下文标记。以后可以用 "'" 或者 "``" 跳转到这个位置 (不移动光标，这不是动作命令)。

m[*m]*

m[或 m] 设置 |'[| 或者 |']| 标记。可以用在执行多个命令的操作符

的定义中。 (不移动光标，这不是动作命令)。

m< *m>*

m< 或 m> 设置 |'<| 或者 |'>| 标记。可以用在改变 `gv` 命令选择的

范围。(不移动光标，这不是动作命令)。

注意 不能设置可视模式，只能设置开始和结束的位置。

:ma *:mark* *E191*

:[range]ma[rk] {a-zA-Z'}

把位置标记 {a-zA-Z'} 设在 [range] 的最后一行，第 0

列。缺省的 [range] 是当前行。

:k

:[range]k{a-zA-Z'} 和 :mark 相同， 但是标记名之前的空格可以省略。

' *'a* *`* *`a*

'{a-z} `{a-z} 跳转到当前缓冲区的位置标记 {a-z}。

'A *'0* *`A* *`0*

'{A-Z0-9} `{A-Z0-9} 跳转到设置位置标记 {A-Z0-9} 的文件所在的标记位置（如果

切换到另外一个文件，这就不是动作命令） {Vi 无此功能}

g' *g'a* *g`* *g`a*

g'{mark} g`{mark}

跳转到指定的位置标记 {mark}，但在当前缓冲区内跳转时，

不改变跳转表。示例: >

g`"

< 跳转到当前文件最近的位置。参看

$VIMRUNTIME/vimrc_example.vim。另见 |:keepjumps|。

{Vi 无此功能}

:marks

:marks 列出所有的位置标记 (这不是动作命令)，

但不包括 |'(|，|')|，|'{| 和 |'}| 标记。

第一列的编号为零。

{Vi 无此功能}

E283

:marks {arg} 列出所有 {arg} 包含的位置标记 (这不是动作命令)。例

如， >

:marks aB

< 列出位置标记 'a' 和 'B'。 {Vi 无此功能}

:delm *:delmarks*

:delm[arks] {marks} 删除指定的位置标记。可以删除的位置标记也包括 A-Z 或

0-9。不能删除 ' 位置标记。

指定的方式包括给出位置标记名的列表和使用连字符分隔的范

围。忽略空格。例如: >

:delmarks a 删除位置标记 a

:delmarks a b 1 删除位置标记 a，b 和 1

:delmarks Aa 删除位置标记 A 和 a

:delmarks p-z 删除位置标记 p 到 z

:delmarks ^.[] 删除位置标记 ^ . [ ]

:delmarks \" 删除位置标记 "

< {Vi 无此功能}

:delm[arks]!　　删除当前缓冲区所有的位置标记，不包括 A-Z 或 0-9 位置标记。

{Vi 无此功能}

位置标记在任何情况下都是不可见的。它只是文件中一个被记住的位置。不要和命名的寄存器混淆，两者毫不相干。

'a - 'z　　小写位置标记，在每个文件内有效。

'A - 'Z　　大写位置标记，也叫做文件标记，在文件间都有效。

'0 - '9　　数字位置标记，在 .viminfo 文件里设置。

只要文件还在缓冲区列表里，小写位置标记 'a 到 'z 就被记住。换而言之，如果在缓冲区列表里删除一个文件，它的位置标记就消失了。如果删除包含某个位置标记的文本行，这个位置标记也就随之消失。

小写位置标记可以和操作符合并使用。例如，"d't" 删除从光标位置到包含 't 标记的文本行。提示: 用 't' 标记代表顶部 (Top)，'b' 标记代表底部 (Bottom)，等等。小写位置标记在撤销/重做时会被复原。

大写位置标记 'A 到 'Z 包含了所在的文件名。 {Vi: 没有大写位置标记} 可以用这些标记在文件间跳转。要和操作符合并使用的大写位置标记必须在当前文件里。即使插入/删除一些行或者同时编辑别的文件，这种标记的行号总是正确的。如果 'viminfo' 选项不为空，大写位置标记由 .viminfo 文件保存。 参阅 |viminfo-file-marks|。

数字位置标记 '0 到 '9 很不一样。它们不是直接设置，而只能从 viminfo 文件

|viminfo-file| 中获取。简单的说，'0 是你上次离开 Vim 时的光标位置。'1 是再上一次，等等。'viminfo' 的 'r' 标志可以指定不记录数字位置标记的文件。参见 |viminfo-file-marks|。

'[*`[*

'[`[到上次改变或者抽出的文本的第一个字符。 {Vi 无此功能}

'] *`]*

'] `] 到上次改变或者抽出文本的最后一个字符。 {Vi 无此功能}

执行完一个操作符后，光标放在操作文本的开始。执行完一个放置命令（"p" 或者 "P"），光标有时放在第一个被插入的位置，有时放在最后一个。上述四个命令可以把光标放在两端的任何一端。例如，在抽出 10 行后，你想要到跳转到最后一行: "10Y']"。在用 "p" 插入数行后你想跳到最底下的那行: "p']"。这对插入的文本也可以。

请 注意: 在删除文本后，除非是面向列块的可视模式，开始和结束的位置是重合的。而如果没有任何改变，这四个命令不会有任何作用。

'< *`<*

'< `< 到上次当前缓冲区选择的可视区域首行或第一个字符。对于列块模式而言，可能也是第一行的最后一个字符（为了能定义列块)。 {Vi 无此功能}

'> *`>*

'> `> 到上次当前缓冲区选择的可视区域末行或最后一个字符。对于列块模式而言，可能也是末行的第一个字符（为了能定义列块)。注意 这里适用 'selection'，该位置可能是刚刚在可视

区域之后的位置。 {Vi 无此功能}

'' *``*

'' ``　　到上次跳转的位置，或者上次 "m'" 或者 "m`" 命令给出的位置。如果使用了 |:keepjumps| 命令修饰符，这个前次上下文标记不会被改变。也可参考 |restore-position|。

'quote *`quote*

'" `"　　到上次离开本缓冲区时的光标位置。缺省是第一行的第一个字符。 参见 |last-position-jump| 以了解如何对所有刚打开的文件使用这个标记。每个缓冲区，而不是每个窗口，只有一个记住的位置。只要这个缓冲区在某一个窗口可见，这个标记是不会改变的。 {Vi 无此功能}

*'^* *`^*

'^ `^　　到上次插入模式关闭的光标位置。|gi| 命令用到这个标记。如果使用了 |:keepjumps| 命令修饰符，这个标记不会改变。{Vi 无此功能}

'. *`.*

'. `.　　到上次改变的位置。这个位置位于或者接近于改变开始的地方。有些命令引起几个改变，那么这个位置应该是接近于这个命令最后改变的地方。例如，插入一个单词的时候，这个位置是在最后一个的字符上。{Vi 无此功能}

'(*`(*

'(`(到当前句子的开始处，就像 |(| 命令。 {Vi 无此功能}

') *`)*

') `) 到当前句子的结尾处，就像 |)| 命令。 {Vi 无此功能}

'{ *`{*

'{ `{ 到当前段落的开始处，就像 |{| 命令。 {Vi 无此功能}

'} *`}*

'} `} 到当前段落的结尾处，就像 |}| 命令。 {Vi 无此功能}

这些命令不是标记本身，而是实现对某个标记的跳转:

]'

]' [count] 次到当前行之后下一个包含小写位置标记的行的第一个非空白字符。 {Vi 无此功能}

]`

]` [count] 次下一个小写位置标记。 {Vi 无此功能}

['

[' [count] 次到当前行之前上一个包含小写位置标记的行的第一个非空白字符。 {Vi 无此功能}

[`

[` [count] 次上一个小写位置标记。 {Vi 无此功能}

:loc[kmarks] {command} *:loc* *:lockmarks*

执行命令 {command}，并且不调整位置标记。这可以用于不影响行的数目的文本改变。如果行数发生变化，那么改变之后的

标记将仍然保持它们的行号，从而实际上移到了别的行上。

以下各项对于插入/删除行不会被调整:

- 小写位置标记 'a - 'z
- 大写位置标记 'A - 'Z
- 数字位置标记 '0 - '9
- 上次插入位置 |'^|
- 上次改变位置 |'.|
- 可视选择区域 |'<| 和 |'>|
- 已设定的标号 |signs| 的行号
- |quickfix| 位置的行号
- 跳转表 |jumplist| 里的位置
- 标签堆栈 |tagstack| 里的位置

以下各项则总会被调整:

- 前次上下文标记 |''|
- 光标位置
- 窗口或者缓冲区的视图
- 折叠 |folding|
- 差异视图 |diff|

:kee[pmarks] {command} *:kee* *:keepmarks*

目前，这只对过滤命令 |:range!| 有效:

- 如果过滤后的行数不少于过滤之前，那么所有的过滤行中的标记保持行号不变。

- 如果行数减少，那么消失了的行中的标记就被删除掉。

在任何情况下，过滤文本之后的标记需要调整行号，以保持和

文本同步。

如果 'cpoptions' 中没有 'R' 标志，那么直接执行过滤命令

和使用 ":keepmarks" 的效果一样。

:keepj *:keepjumps*
:keepj[umps] {command}

在 {command} 的移动不改变 |''|，|'.| 和 |'^| 标记，跳

转表 |jumplist| 或是改变表 |changelist|。这对于自动改

变或者插入文本而不需要用户跳转到那个位置有用。例如，当

更新 "最近改变" 的时间标签: >

```
:let lnum = line(".")
:keepjumps normal gg
:call SetLastChange()
:keepjumps exe "normal " . lnum . "G"
```
<

注意 ":keepjumps" 必须在每个命令里使用。如果在命令里调

用了函数，则该函数里的命令仍然可能改变跳转表。另外，

":keepjumps exe 'command '" 里的 "command" 不会保护跳

转表。应该用: ":exe 'keepjumps command'"

==

8. 跳转 *jump-motions*

"跳转" 包括如下的命令: "'"、"`"、"G"、"/"、"?"、"n"、"N"、"%"、"("、")"、

"[["、"]]"、"{"、"}"、":s"、":tag"、"L"、"M"、"H" 和开始编辑新文件的命令。如果

用这些命令使光标 "跳转"，那么跳转之前的光标位置会被记住。除非包含该位置的行被改变或者删除，你可以用 "''" 和 "``" 命令返回这个位置。

CTRL-O

CTRL-O 转到跳转表里第 [count] 个较旧的光标位置

(不是动作命令). {Vi 无此功能}

{仅当编译时加入 |+jumplist| 特性才有效}

<Tab> 或 *CTRL-I* *<Tab>*

CTRL-I 转到跳转表里第 [count] 个较新的光标位置

(不是动作命令)。

{Vi 无此功能}

{仅当编译时加入 |+jumplist| 特性才有效}

:ju *:jumps*

:ju[mps] 打出跳转表 (不是动作命令)。 {Vi 无此功能}

{仅当编译时加入 |+jumplist| 特性才有效}

jumplist

跳转表用来记住跳转的位置。用 CTRL-O 和 CTRL-I 命令，你可以跳回到较早前的跳转位置，然后在跳回来。这样，你就可以在这个列表上下移动。每个窗口有独立的跳转表，每个表最多能存储 100 项。

{仅当编译时加入 |+jumplist| 特性才有效}

例如，在三个跳转命令之后你有如下的跳转表:

```
  jump line    col file/text ~
```

```
    3       1      0 some text ~
    2      70      0 another line ~
    1    1154     23 end. ~
>~
```

"file/text" 列显示文件名，如果是当前文件，则显示跳转所在的文本（为了能在窗口里显示，去掉开头的缩排空白并截断过长的行)。

你现在位于第 1167 行。如果你用 CTRL-O 命令，光标会到第 1154 行。结果如下:

```
  jump line    col file/text ~
    2       1      0 some text ~
    1      70      0 another line ~
>   0    1154     23 end. ~
    1    1167      0 foo bar ~
```

指针会指向上一次跳转的位置。下一个 CTRL-O 命令会使用更上面的那项。而下一个 CTRL-I 命令则会使用下面的那项。如果指针在最后一项之下，那说明你还没用过 CTRL-I 或 CTRL-O 命令。此时，CTRL-O 命令会把光标位置加到跳转表里，这样你以后就可以回到用 CTRL-O 之前的位置。在本例中，这是第 1167 行。

更多的 CTRL-O 命令会使你分别到第 70 和第 1 行。如果你用 CTRL-I，你又可以回到第
1154 和第 1167 行。 注意: "jump" 列的数字指示你用 CTRL-O 或 CTRL-I 到该位置所需的次数。

如果你用跳转命令，当前的行号被插到跳转表的最后。如果相同的行已经在跳转表里，那会被删除。结果是，CTRL-O 就会直接回到该行之前的位置。

如果用了 |:keepjumps| 命令修饰符，跳转就不会被保存在跳转表里。一些其它场合也不

保存跳转，例如在 |:global| 命令的里面。你可以用 "m'" 设置 ' 位置标记来显式加入跳转。注意 调用 setpos() 做不到这一点。

在 CTRL-O 命令之后，你到了第 1154 行。如果你给出另外一个跳转命令（比如，"G"），那么跳转表就会成为:

```
  jump line    col file/text ~
     4       1      0 一些文字 ~
     3      70      0 另外一行 ~
     2    1167      0 foo bar ~
     1    1154     23 end. ~
 > ~
```

删除和插入行以后，行号会得到调整。不过，如果你不保存文件而放弃编辑，例如 ":n!"，该调整会失败。

如果你分割一个窗口，跳转表会复制到新的窗口里。

如果在 'viminfo' 选项里有 ' 项，跳转表会保存在 viminfo 文件里。这样，重新启动 Vim 就会恢复跳转表。

改 变 表 跳 转 *changelist* *change-list-jumps* *E664*

在改变文本以后，光标的位置被记住。每个改变都会记住一个位置，从而可以使该操作能被撤销。除非，这个位置和上一个改变很接近。可以用两个命令跳转到改变所在的位置，包括那些已被撤销的:

g; *E662*

g;　　转到改变表里第［count］个较旧的位置。

如果［count］比所有的较旧的改变位置都多，回到最老的那

个。

如果没有较旧的改变，给出一个错误消息。

(不是动作命令)

{Vi 无此功能}

{仅当编译时加入 |+jumplist| 特性才有效}

g, *E663*

g,　　转到改变表里第［count］个较新的位置。

和 |g;| 类似，但朝向相反的方向。

(不是动作命令)

{Vi 无此功能}

{仅当编译时加入 |+jumplist| 特性才有效}

使用次数的时候你向后或向前跳转到尽可能接近该次数的位置。这样，你可以用 "999g;" 跳到还能记住的最初的改变所在的位置。改变的最大项目数和跳转表 |jumplist| 一致。

当两个可以撤销的改变在同一行，并且所在的列差小于 'textwidth'，只有后一个会被记住。这可以避免同一行里一系列很小的改变，例如 "xxxxx"，在改变表里占据很多位置。如果 'textwidth' 为 0，则使用 'wrapmargin'。如果它也没有设置，就用固定的数目 79。细节：考虑到速度的影响，计算用的是字节数而不是字符数（这只对多字节编码有意义)。

请 注意 如果后来有插入和删除文本，尤其删除整行的时候，光标位置和当时改变的位置

可能稍有不同。

如果用到 |:keepjumps| 命令修饰符，改变的位置不会被记住。

:changes

:changes 打出改变表。 ">" 字符指示当前的位置。在一个改变之后，它在最新的一项之后，这意味着 "g;" 会带你到最新一项的位置。第一列则指示要到该位置需要的次数。例如:

```
change line  col text ~
    3       9     8 bla bla bla
    2      11    57 foo is a bar
    1      14    54 最新改变的行
>
```

"3g;" 命令会带你到第 9 行。这时，":changes" 的结果是:

```
change line  col text ~
>   0       9     8 bla bla bla
    1      11    57 foo is a bar
    2      14    54 最新改变的行
```

现在你可以用 "g," 到第 11 行，"2g," 到第 14 行。

==========

9. 其他动作 *various-motions*

%

% 找到本行的光标所在或其后的下一个项目，并跳转到它的匹配。

|inclusive| 闭动作。

项目可以是:

([{}])　　小括号或者（花/方）括号。

　　(这可以用 'matchpairs' 选项改变)

/* */　　C-风格的注释的开始或结尾

#if、#ifdef、#else、#elif、#endif

　　C 预处理条件宏（光标在 # 上，或者其后

　　没有（[{ 的时候）

要匹配其它的项目可以用 matchit 插件，参见

|matchit-install|。此插件也可用于跳过注释中的匹配。

在 'cpoptions' 包含 "M" 时，|cpo-M| 括号之前的反斜杠被忽略。若不然，那么括号之前的反斜杠的数目很重要。带偶数数目的不能匹配带奇数数目的。这样，在 "(\))" 和 "\((\)" 里，第一个和最后一个小括号互相匹配。

如果 'cpoptions' 里没有 '%' 字符 |cpo-%|，那么双引号里的括号被忽略，除非一行里的括号数目不对称，而且该行和前一行不以反斜杠结尾。'('、'{'、'['、']'、'}' 和 ')' 也被忽略（单引号里的括号)。 注意 这对 C 适用，但对 Perl 就不行。Perl 里单引号用来括起字符串。

对注释中的匹配并无特殊处理。可以用 matchit 插件 |matchit-install| 或者用引号括起匹配。

这里不能用计数，{count}% 跳转到文件里百分之 {count} 的

那行 |N%|。

在 #if/#else/#endif 上用 % 使该动作面向行。

[(

[(反向第 [count] 个的未匹配的 '('。

|exclusive| 开动作。{Vi 无此功能}。

[{

[{ 反向第 [count] 个的未匹配的 '{'。

|exclusive| 开动作。{Vi 无此功能}。

])

]) 正向第 [count] 个的未匹配的 ')'。

|exclusive| 开动作。{Vi 无此功能}。

]}

]} 正向第 [count] 个的未匹配的 '}'。

|exclusive| 开动作。{Vi 无此功能}。

以上四个命令用于转到当前代码块的开始或者结尾位置。这和在 '('、')'、'{' 或 '}'

上用 "%" 类似，但你可以在代码块的任何位置这么做，这对 C 程序很有用。例如: 在

"case x:" 上用 "[{" 会把你带回到 switch 语句上。

]m

]m 正向第 [count] 个方法（method）的开始处（适用于 Java

或类似结构的语言)。如果不在某个方法开始处之前，则跳转

到类的开始或结束处。如果光标之后已无 '{'，会有一个错

误。

|exclusive| 开动作。{Vi 无此功能}。

]M

]M 正向第 [count] 个方法的结束处（适用于 Java 或类似结构的语言)。如果不在某个方法开始处之前，则跳转到类的开始或结束处。如果光标之后已无 '}'，会有一个错误。

|exclusive| 开动作。{Vi 无此功能}。

[m

[m 反向第 [count] 个方法的开始处（适用于 Java 或类似结构的语言)。如果不在某个方法开始处之前，则跳转到类的开始或结束处。如果光标之前已无 '{'，会有一个错误。

|exclusive| 开动作。{Vi 无此功能}。

[M

[M 反向第 [count] 个方法的结束处（适用于 Java 或类似结构的语言)。如果不在某个方法开始处之前，则跳转到类的开始或结束处。如果光标之前已无 '}'，会有一个错误。

|exclusive| 开动作。{Vi 无此功能}。

以上这些命令假设文件包含一个有方法的类定义。类定义用 '{' 和 '}' 包围，而方法定义亦然。Java 语言便是如此。源文件看起来应像这样: >

```
// 注释
class foo {
    int method_one() {
        body_one();
    }
    int method_two() {
```

```
            body_two();
        }
    }
```

开始，光标在 "body_two()" 上，用 "[m" 会跳转到 "method_two()" 开始的那个 '{' (显然，如果方法定义很长，这就更有用！)。用 "2[m" 会跳转到 "method_one()" 的开始处。用 "3[m" 会跳转到类的开始处。

[#

[# 反向第 [count] 个未匹配的 "#if" 或 "#else"。

|exclusive| 开动作。{Vi 无此功能}。

]#

]# 正向第 [count] 个未匹配的 "#else" 或 "#endif"。

|exclusive| 开动作。{Vi 无此功能}。

这两个命令对有 #if/#else/#endif 结构的 C 程序很有用。它们把你带到当前行所在的 #if/#else/#endif 的开始或结束处。然后，你可以用 "%" 找到相匹配的行。

[star *[/*

[* 或 [/ 反向第 [count] 个 C 注释的开始 "/*"。

|exclusive| 开动作。{Vi 无此功能}。

]star *]/*

]* 或]/ 正向第 [count] 个 C 注释的结束 "*/"。

|exclusive| 开动作。{Vi 无此功能}。

H

H 到窗口从顶部（Home）算第 [count] 行（缺省：窗口的第一行）并停在第一个非空白字符上。|linewise| 行动作。参见

'startofline' 选项。光标还要根据 'scrolloff' 调整。

M

M 到窗口的中间（Middle）一行并停在第一个非空白字符。

|linewise| 行动作。参见 'startofline' 选项。

L

L 到窗口从底部（Last）算第 [count] 行（缺省：窗口的最后一行）并停在第一个非空白字符上。|linewise| 行动作。参见 'startofline' 选项。光标还要根据 'scrolloff' 调整。

<LeftMouse> 到屏幕上鼠标点击的位置。|exclusive| 开动作。参见 |<LeftMouse>|。如果鼠标在状态行上，则所属的窗口被激活但光标位置不改变。{Vi 无此功能}

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