Vim 从入门到精通

English | Japanese | Portuguese | Russian

chat on gitter

简介

什么是 Vim?

Vim 是一个历史悠久的文本编辑器，可以追溯到
qed。
Bram Moolenaar 于
1991 年发布初始版本。

Linux、Mac 用户，可以使用包管理器安装 Vim，对于 Windows 用户，可以从
我的网盘 下载。
该版本可轻易添加 `python` 、 `python3` 、 `lua` 等支持，只需要安装
python、lua
即可。

项目在 Github 上开发，项目讨论请订阅
`vim_dev` 邮件列表。

通过阅读 Why, oh WHY, do those #?@! nutheads use vi?
来对 Vim 进行大致的了解。

Vim 哲学

Vim 采用模式编辑的理念，即它提供了多种模式，按键在不同的模式下作用不同。
你可以在**普通模式** 下浏览文件，在**插入模式**下插入文本，
在**可视模式**下选择行，在**命令模式**下执行命令等等。起初这听起来可能很复杂，
但是这有一个很大的优点：不需要通过同时按住多个键来完成操作，
大多数时候你只需要依次按下这些按键即可。越常用的操作，所需要的按键数量越少。

和模式编辑紧密相连的概念是 **操作符** 和 **动作**。**操作符** 指的是开始某个行为，
例如：修改、删除或者选择文本，之后你要用一个 **动作** 来指定需要操作的文本区域。
比如，要改变括号内的文本，需要执行 `ci(` （读做 `change inner parentheses`）；
删除整个段落的内容，需要执行 `dap` （读做：`delete around paragraph`）。

如果你能看见 Vim 老司机操作，你会发现他们使用 Vim 脚本语言就如同钢琴师弹钢琴一样。复杂的操作只需要几个按键就能完成。他们甚至不用刻意去想，因为这已经成为肌肉记忆了。这减少认识负荷并帮助人们专注于实际任务。

入门

Vim 自带一个交互式的教程，内含你需要了解的最基础的信息，你可以通过终端运行以下命令打开教程：

```
$ vimtutor
```

不要因为这个看上去很无聊而跳过，按照此教程多练习。你以前用的 IDE 或者其他编辑器很少是有“模式”概念的，因此一开始你会很难适应模式切换。但是你 Vim 使用的越多，肌肉记忆 将越容易形成。

Vim 基于一个 vi 克隆，叫做 Stevie，支持两种运行模式："compatible" 和 "nocompatible"。在兼容模式下运行 Vim 意味着使用 vi 的默认设置，而不是 Vim 的默认设置。除非你新建一个用户的 `vimrc` 或者使用 `vim -N` 命令启动 Vim，否则就是在兼容模式下运行 Vim！请大家不要在兼容模式下运行 Vim。

下一步

1. 创建你自己的 vimrc。
2. 在第一周准备备忘录。
3. 通读基础章节了解 Vim 还有哪些功能。
4. 按需学习！Vim 是学不完的。如果你遇到了问题，先上网寻找解决方案，你的问题可能已经被解决了。Vim 拥有大量的参考文档，知道如何利用这些参考文档很有必要：获取离线帮助。
5. 浏览附加资源。

最后一个建议：使用插件之前，请先掌握 Vim 的基本操作。很多插件都只是对 Vim 自带功能的封装。

返回主目录 ⤴

精简的 vimrc

Vim 启动是会按照一定的优先顺序来搜索配置文件，这个顺序，可以通过 `:version` 命令查看。下面分 Windows 系统，
和 *niux 系统分别来说明 Vim 是如何载入配置文件的。

Windows 系统

```
   system vimrc file: "$VIM\vimrc"
     user vimrc file: "$HOME\_vimrc"
 2nd user vimrc file: "$HOME\vimfiles\vimrc"
 3rd user vimrc file: "$VIM\_vimrc"
      user exrc file: "$HOME\_exrc"
  2nd user exrc file: "$VIM\_exrc"
  system gvimrc file: "$VIM\gvimrc"
    user gvimrc file: "$HOME\_gvimrc"
2nd user gvimrc file: "$HOME\vimfiles\gvimrc"
3rd user gvimrc file: "$VIM\_gvimrc"
       defaults file: "$VIMRUNTIME\defaults.vim"
    system menu file: "$VIMRUNTIME\menu.vim"
```

我们只看上面这一段，Vim 会优先读取 user vimrc file: `$HOME\_vimrc`, 当这一文件不存在是，
Vim 再去寻找 2nd user vimrc file: `$HOME\vimfiles\vimrc`; 倘若这个文件还是不存在，那么 Vim
会去继续寻找 3rd user vimrc file: `$VIM\_vimrc`。 了解以上顺序后，就不会再因为 Vim
总是不读取配置文件而感到烦恼了。

Linux 或者 Mac OS

同 Windows 系统类似，也可以使用 `:version` 命令查看 vim 载入配置的优先顺序。

```
     系统 vimrc 文件: "/etc/vimrc"
     用户 vimrc 文件: "$HOME/.vimrc"
第二用户 vimrc 文件: "~/.vim/vimrc"
      用户 exrc 文件: "$HOME/.exrc"
       defaults file: "$VIMRUNTIME/defaults.vim"
         $VIM 预设值: "/etc"
 $VIMRUNTIME 预设值: "/usr/share/vim/vim81"
```

你可以在网上找到许多精简的 vimrc 配置文件，我的版本可能并不是最简单的版本，但是我的版本提供了一套我认为良好的，非常适合入门的设置。

最终你需要阅读完那些设置，然后自行决定需要使用哪些。:-)

精简的 vimrc 地址：minimal-vimrc

如果你有兴趣，这里是我（原作者）的 vimrc。

建议：大多数插件作者都维护不止一个插件并且将他们的 vimrc 放在 Github 上展示（通常放在叫做 "vim-config" 或者 "dotfiles" 的仓库中），所以当你发现你喜欢的插件时，去插件维护者的 Github 主页看看有没有这样的仓库。

返回主目录 ⤴

我正在使用什么样的 Vim

使用 `:version` 命令将向你展示当前正在运行的 Vim 的所有相关信息，包括它是如何编译的。

第一行告诉你这个二进制文件的编译时间和版本号，比如：7.4。接下来的一行呈现 `Included patches: 1-1051`，这是补丁版本包。因此你 Vim 确切的版本号是 7.4.1051。

另一行显示着一些像 `Tiny version without GUI` 或者 `Huge version with GUI` 的信息。很显然这些信息告诉你当前的 Vim 是否支持 GUI，例如：从终端中运行 `gvim` 或者从终端模拟器中的 Vim 内运行 `:gui` 命令。另一个重要的信息是 `Tiny` 和 `Huge`。Vim 的特性集区分被叫做 `tiny`，`small`，`normal`，`big` and `huge`，所有的都实现不同的功能子集。

`:version` 主要的输出内容是特性列表。`+clipboard` 意味这剪贴板功能被编译支持了，`-clipboard` 意味着剪贴板特性没有被编译支持。

一些功能特性需要编译支持才能正常工作。例如：为了让 `:prof` 工作，你需要使用 `huge` 模式编译的 Vim，因为那种模式启用了 `+profile` 特性。

如果你的输出情况并不是那样，并且你是从包管理器安装 Vim 的，确保你安装了 `vim-x`，`vim-x11`，`vim-gtk`，`vim-gnome` 这些包或者相似的，因为这些包通常都是 `huge` 模式编译的。

你也可以运行下面这段代码来测试 Vim 版本以及功能支持：

```
" Do something if running at least Vim 7.4.42 with
+profile enabled.
if (v:version > 704 || v:version == 704 &&
has('patch42')) && has('profile')
  " do stuff
endif
```

相关帮助：

```
:h :version
:h feature-list
:h +feature-list
:h has-patch
```

返回主目录 ⤴

备忘录

为了避免版权问题，我只贴出链接：

- http://people.csail.mit.edu/vgod/vim/vim-cheat-sheet-en.png
- https://cdn.shopify.com/s/files/1/0165/4168/files/preview.png
- http://www.nathael.org/Data/vi-vim-cheat-sheet.svg
- http://michael.peopleofhonoronly.com/vim/vim_cheat_sheet_for_programmers_screen.png
- http://www.rosipov.com/images/posts/vim-movement-commands-cheatsheet.png

或者在 Vim 中快速打开备忘录：vim-cheat40。

返回主目录 ⤴

基础

缓冲区，窗口，标签

Vim 是一个文本编辑器。每次文本都是作为**缓冲区**的一部分显示的。每一份文件都是在他们自己独有的缓冲区打开的，插件显示的内容也在它们自己的缓冲区中。

缓冲区有很多属性，比如这个缓冲区的内容是否可以修改，或者这个缓冲区是否和文件相关联，是否需要同步保存到磁盘上。

窗口 是缓冲区上一层的视窗。如果你想同时查看几个文件或者查看同一文件的不同位置，那样你会需要窗口。

请别把他们叫做 *分屏* 。你可以把一个窗口分割成两个，但是这并没有让这两个窗口完全 *分离* 。

窗口可以水平或者竖直分割并且现有窗口的高度和宽度都是可以被调节设置的，因此，如果你需要多种窗口布局，请考虑使用标签。

标签页 （标签）是窗口的集合。因此当你想使用多种窗口布局时候请使用标签。

简单的说，如果你启动 Vim 的时候没有附带任何参数，你会得到一个包含着一个呈现一个缓冲区的窗口的标签。

顺带提一下，缓冲区列表是全局可见的，你可以在任何标签中访问任何一个缓冲区。

返回主目录 ⤴

已激活、已载入、已列出、已命名的缓冲区

用类似 `vim file1` 的命令启动 Vim 。这个文件的内容将会被加载到缓冲区中，你现在有一个**已载入的缓冲区**。如果你在 Vim 中保存这个文件，缓冲区内容将会被同步到磁盘上（写回文件中）。

由于这个缓冲区也在一个窗口上显示，所以他也是一个**已激活的缓冲区**。如果你现在通过 `:e file2` 命令加载另一个文件，`file1` 将会变成一个**隐藏的缓冲区**，并且 `file2` 变成已激活缓冲区。

使用 `:ls` 我们能够列出所有可以列出的缓冲区。插件缓冲区和帮助缓冲区通常被标记为不可以列出的缓冲区，因为那并不是你经常需要在编辑器中编辑的常规文件。通过 `:ls!` 命令可以显示被放入缓冲区列表的和未被放入列表的缓冲区。

未命名的缓冲区是一种没有关联特定文件的缓冲区，这种缓冲区经常被插件使用。比如 `:enew` 将会创建一个无名临时缓冲区。添加一些文本然后使用 `:w /tmp/foo` 将他写入到磁盘，这样这个缓冲区就会变成一个**已命名的缓冲区**。

返回主目录 ⤴

参数列表

全局缓冲区列表是 Vim 的特性。在这之前的 vi 中，仅仅只有参数列表，参数列表在 Vim 中依旧可以使用。

每一个通过 shell 命令传递给 Vim 的文件名都被记录在一个参数列表中。可以有多个参数列表：默认情况下所有参数都被放在全局参数列表下，但是你可以使用 `:arglocal` 命令去创建一个新的本地窗口的参数列表。

使用 `:args` 命令可以列出当前参数。使用 `:next`，`:previous`，`:first`，`:last` 命令可以在切换在参数列表中的文件。通过使用 `:argadd`，`:argdelete` 或者 `:args` 等命令加上一个文件列表可以改变参数列表。

偏爱缓冲区列表还是参数列表完全是个人选择，我的印象中大多数人都是使用缓冲区列表的。

然而参数列表在有些情况下被大量使用：批处理
使用 `:argdo`！ 一个简单的重构例子：

```
:args **/*.[ch]
:argdo %s/foo/bar/ge | update
```

这条命令将替换掉当前目录下以及当前目录的子目录中所有的 C 源文件和头文件中的“foo”，并用“bar”代替。

相关帮助：`:h argument-list`

返回主目录 ⤴

按键映射

使用 `:map` 命令家族你可以定义属于你自己的快捷键。该家族的每一个命令都限定在特定的模式下。从技术上来说 Vim 自带高达 12 中模式，其中 6 种可以被映射。另外一些命令作用于多种模式：

递归	非递归	模式
`:map`	`:noremap`	normal, visual, operator-pending
`:nmap`	`:nnoremap`	normal
`:xmap`	`:xnoremap`	visual
`:cmap`	`:cnoremap`	command-line
`:omap`	`:onoremap`	operator-pending
`:imap`	`:inoremap`	insert

例如：这个自定义的快捷键只在普通模式下工作。

```
:nmap <space> :echo "foo"<cr>
```

使用 `:nunmap <space>` 可以取消这个映射。

对于更少数，不常见的模式（或者他们的组合），查看 `:h map-modes`。

到现在为止还好，对新手而言有一个问题会困扰他们：`:nmap` 是**递归执行**的！结果是，右边执行可能的映射。

你自定义了一个简单的映射去输出“Foo”：

```
:nmap b :echo "Foo"<cr>
```

但是如果你想要映射 `b` （回退一个单词）的默认功能到一个键上呢？

```
:nmap a b
```

如果你敲击 a，我们期望着光标回退到上一个单词，但是实际情况是“Foo”被输出到命令行里！因为在右边，`b` 已经被映射到别的行为上了，换句话说就是 `:echo "Foo"<cr>`。

解决此问题的正确方法是使用一种 *非递归* 的映射代替：

```
:nnoremap a b
```

经验法则：除递归映射是必须的，否则总是使用非递归映射。

通过不给一个右值来检查你的映射。比如 `:nmap` 显示所以普通模式下的映射，`:nmap <leader>` 显示所有以 `<leader>` 键开头的普通模式下的映射。

如果你想禁止用标准映射，把他们映射到特殊字符 `<nop>` 上，例如：`:noremap <left> <nop>`。

相关帮助：

```
:h key-notation
:h mapping
:h 05.3
```

返回主目录 ⤴

映射前置键

映射前置键（Leader 键）本身就是一个按键映射，默认为 `\`。我们可以通过在 `map` 中调用 `<leader>` 来为把它添加到其他按键映射中。

```
nnoremap <leader>h :helpgrep<space>
```

这样，我们只需要先按 `\` 然后按 `h` 就可以激活这个映射 `:helpgrep<space>`。如果你想通过先按 `空格` 键来触发，只需要这样做：

```
let g:mapleader = ' '
nnoremap <leader>h :helpgrep<space>
```

此处建议使用 `g:mapleader`，因为在 Vim 脚本中，函数外的变量缺省的作用域是全局变量，但是在函数内缺省作用域是局部变量，而设置快捷键前缀需要修改全局变量 `g:mapleader` 的值。

另外，还有一个叫 `<localleader>` 的，可以把它理解为局部环境中的 `<leader>`，默认值依然为 `\`。当我们需要只对某一个条件下（比如，特定文件类型的插件）的缓冲区设置特别的 `<leader>` 键，那么我们就可以通过修改当前环境下的 `<localleader>` 来实现。

注意：如果你打算设置 Leader 键，请确保在设置按键映射之前，先设置好 Leader 键。如果你先设置了含有 Leader 键的映射，然后又修改了 Leader 键，那么之前映射内的 Leader 键是不会因此而改变的。你可以通过执行 `:nmap <leader>` 来查看普通模式中已绑定给 Leader 键的所有映射。

请参阅 `:h mapleader` 与 `:h maploacalleader` 来获取更多帮助。

返回主目录 ⤴

寄存器

寄存器就是存储文本的地方。我们常用的「复制」操作就是把文本存储到寄存器，「粘贴」操作就是把文本从寄存器中读出来。顺便，在 Vim 中复制的快捷键是 `y`，粘贴的快捷键是 `p`。

Vim 为我们提供了如下的寄存器：

类型	标识	读写者	是否为只读	包含的字符来源
Unnamed	`"`	vim	否	最近一次的复制或删除操作 (`d`, `c`, `s`, `x`, `y`)
Numbered	`0`至`9`	vim	否	寄存器 `0`: 最近一次复制。寄存器 `1`: 最近一次删除。寄存器 `2`: 倒数第二次删除，以此类推。对于寄存器 `1` 至 `9`，他们其实是只读的最多包含 9 个元素的队列。这里的队列即为数据类型 queue
Small delete	`-`	vim	否	最近一次行内删除
Named	`a`至`z`, `A`至`Z`	用户	否	如果你通过复制操作存储文本至寄存器 `a`，那么 `a` 中的文本就会被完全覆盖。如果你存储至 `A`，那么会将文本添加给寄存器 `a`，不会覆盖之前已有的文本
Read-only	`:`与`.`和`%`	vim	是	`:`: 最近一次使用的命令，`.`: 最近一次添加的文本，`%`: 当前的文件名
Alternate buffer	`#`	vim	否	大部分情况下，这个寄存器是当前窗口中，上一次访问的缓冲区。请参阅 `:h alternate-file` 来获取更多帮助

类型	标识	读写者	是否为只读	包含的字符来源
Expression	`=`	用户	否	复制 VimL 代码时，这个寄存器用于存储代码片段的执行结果。比如，在插入模式下复制 `<c-r>=5+5<cr>`，那么这个寄存器就会存入 10
Selection	`+`和`*`	vim	否	`*` 和 `+` 是 剪贴板 寄存器
Drop	`~`	vim	是	最后一次拖拽添加至 Vim 的文本（需要 "+dnd" 支持，暂时只支持 GTK GUI。请参阅 `:help dnd` 及 `:help quote~`）
Black hole	`_`	vim	否	一般称为黑洞寄存器。对于当前操作，如果你不希望在其他寄存器中保留文本，那就在命令前加上 `_`。比如，`"_dd` 命令不会将文本放到寄存器 `"`、`1`、`+` 或 `*` 中
Last search pattern	`/`	vim	否	最近一次通过 `/`、`?` 或 `:global` 等命令调用的匹配条件

只要不是只读的寄存器，用户都有权限修改它的内容，比如：

```
:let @/ = 'register'
```

这样，我们按 n 的时候就会跳转到单词"register" 出现的地方。

有些时候，你的操作可能已经修改了寄存器，而你没有察觉到。请参阅 `:h registers` 获取更多帮助。

上面提到过，复制的命令是 `y`，粘贴的命令是 `p` 或者 `P`。但请注意，Vim 会区分「字符选取」与「行选取」。请参阅 `:h linewise` 获取更多帮助。

行选取:
命令 `yy` 或 `Y` 都是复制当前行。这时移动光标至其他位置，按下 `p` 就可以在光标下方粘贴复制的行，按下 `P` 就可以在光标上方粘贴至复制的行。

字符选取:
命令 `0yw` 可以复制第一个单词。这时移动光标至其他位置，按下 `p` 就可以在当前行、光标后的位置粘贴单词，按下 `P` 就可以在当前行、光标前的位置粘贴单词。

将文本存到指定的寄存器中:
命令 `"aY` 可以将当前行复制，并存储到寄存器 `a` 中。这时移动光标至其他位置，通过命令 `"AY` 就可以把这一行的内容扩展到寄存器 `a` 中，而之前存储的内容也不会丢失。

为了便于理解和记忆，建议大家现在就试一试上面提到的这些操作。操作过程中，你可以随时通过 `:reg` 来查看寄存器的变化。

有趣的是:
在 Vim 中，`y` 是复制命令，源于单词 "yanking"。而在 Emacs 中，"yanking" 代表的是粘贴（或者说，重新插入刚才删掉的内容），而并不是复制。

返回主目录 ⤴

范围

范围 (Ranges) 其实很好理解，但很多 Vim 用户的理解不到位。

- 很多命令都可以加一个数字，用于指明操作范围
- 范围可以是一个行号，用于指定某一行
- 范围也可以是一对通过 `,` 或 `;` 分割的行号
- 大部分命令，默认只作用于当前行
- 只有 `:write` 和 `:global` 是默认作用于所有行的

范围的使用是十分直观的。以下为一些例子（其中，`:d` 为 `:delete` 的缩写）：

命令	操作的行
`:d`	当前行
`:.d`	当前行
`:1d`	第一行
`:$d`	最后一行
`:1,$d`	所有行
`:%d`	所有行（这是 `1,$` 的语法糖）
`:.,5d`	当前行至第 5 行
`:,5d`	同样是当前行至第 5 行
`:,+3d`	当前行及接下来的 3 行
`:1,+3d`	第一行至当前行再加 3 行
`:,-3d`	当前行及向上的 3 行（Vim 会弹出提示信息，因为这是一个保留的范围）
`:3,'xdelete`	第三行至标注 为 x 的那一行
`:/^foo/,$delete`	当前行以下，以字符 "foo" 开头的那一行至结尾
`:/^foo/+1,$delete`	当前行以下，以字符 "foo" 开头的那一行的下一行至结尾

需要注意的是，`;` 也可以用于表示范围。区别在于，`a,b` 的 `b` 是以当前行作为参考的。而 `a;b` 的 `b` 是以 `a` 行作为参考的。举个例子，现在你的光标在第 5 行。这时 `:1,+1d` 会删除第 1 行至第 6 行，而 `:1;+1d` 会删除第 1 行和第 2 行。

如果你想设置多个寻找条件，只需要在条件前加上 `/`，比如：

```
:/foo//bar//quux/d
```

这就会删除当前行之后的某一行。定位方式是，先在当前行之后寻找第一个包含 "foo" 字符的那一行，然后在找到的这一行之后寻找第一个包含 "bar" 字符的那一行，然后再在找到的这一行之后寻找第一个包含 "quux" 的那一行。删除的就是最后找到的这一行。

有时，Vim 会在命令前自动添加范围。举个例子，如果你先通过 `V` 命令进入行选取模式，选中一些行后按下 `:` 进入命令模式，这时候你会发现 Vim 自动添加了 `'<,'>` 范围。这表示，接下来的命令会使用之前选取的行号作为范围。但如果后续命令不支持范围，Vim 就会报错。为了避免这样的情况发生，有些人会设置这样的按键映射：`:vnoremap foo :<c-u>command`，组合键 `Ctrl + u` 可以清除当前命令行中的内容。

另一个例子是在普通模式中按下 `!!`，命令行中会出现 `:.!`。如果这时你如果输入一个外部命令，那么当前行的内容就会被这个外部命令的输出替换。你也可以通过命令 `:?^$?+1,/^$/-1!ls` 把当前段落的内容替换成外部命令 `ls` 的输出，原理是向前和向后各搜索一个空白行，删除这两个空白行之间的内容，并将外部命令 `ls` 的输出放到这两个空白行之间。

请参阅以下两个命令来获取更多帮助：

```
:h cmdline-ranges
:h 10.3
```

返回主目录 ⤴

标注

你可以使用标注功能来标记一个位置，也就是记录文件某行的某个位置。

标注	设置者	使用
`a-z`	用户	仅对当前的一个文件生效，也就意味着只可以在当前文件中跳转
`A-Z`	用户	全局标注，可以作用于不同文件。大写标注也称为「文件标注」。跳转时有可能会切换到另一个缓冲区
`0-9`	viminfo	`0` 代表 viminfo 最后一次被写入的位置。实际使用中，就代表 Vim 进程最后一次结束的位置。`1` 代表 Vim 进程倒数第二次结束的位置，以此类推

如果想跳转到指定的标注，你可以先按下 `'` / `g'` 或者 `` ` `` / `` g` `` 然后按下标注名。

如果你想定义当前文件中的标注，可以先按下 `m` 再按下标注名。比如，按下 `mm` 就可以把当前位置标注为 `m`。在这之后，如果你的光标切换到了文件的其他位置，只需要通过 `'m` 或者 `` `m `` 即可回到刚才标注的行。区别在于，`'m` 会跳转回被标记行的第一个非空字符，而 `` `m `` 会跳转回被标记行的被标记列。根据 viminfo 的设置，你可以在退出 Vim 的时候保留小写字符标注。请参阅 `:h viminfo-'` 来获取更多帮助。

如果你想定义全局的标注，可以先按下 `m` 再按下大写英文字符。比如，按下 `mM` 就可以把当前文件的当前位置标注为 `M`。在这之后，就算你切换到其他的缓冲区，依然可以通过 `'M` 或 `` `M `` 跳转回来。

关于跳转，还有以下的方式：

按键	跳转至
`'[` 与 `` `[ ``	上一次修改或复制的第一行或第一个字符
`']` 与 `` `] ``	上一次修改或复制的最后一行或最后一个字符
`'<` 与 `` `< ``	上一次在可视模式下选取的第一行或第一个字符
`'>` 与 `` `> ``	上一次在可视模式下选取的最后一行或最后一个字符
`''` 与 `` `' ``	上一次跳转之前的光标位置
`'"` 与 `` `" ``	上一次关闭当前缓冲区时的光标位置
`'^` 与 `` `^ ``	上一次插入字符后的光标位置
`'.` 与 `` `. ``	上一次修改文本后的光标位置
`'(` 与 `` `( ``	当前句子的开头
`')` 与 `` `) ``	当前句子的结尾
`'{` 与 `` `{ ``	当前段落的开头
`'}` 与 `` `} ``	当前段落的结尾

标注也可以搭配 范围 一起使用。前面提到过，如果你在可视模式下选取一些文本，然后按下 `:`，这时候你会发现命令行已经被填充了 `:'<,'>`。对照上面的表格，现在你应该明白了，这段代表的就是可视模式下选取的范围。

请使用 `:marks` 命令来显示所有的标注，参阅 `:h mark-motions` 来获取关于标注的更多帮助。

返回主目录 ⤴

补全

Vim 在插入模式中为我们提供了多种补全方案。如果有多个补全结果，Vim 会弹出一个菜单供你选择。

常见的补全有标签、项目中引入的模块或库中的方法名、文件名、字典及当前缓冲区的字段。

针对不同的补全方案，Vim 为我们提供了不同的按键映射。这些映射都是在**插入模式中**通过 `Ctrl` + `x` 来触发：

映射	类型	帮助文档
`<c-x><c-l>`	整行	`:h i^x^l`
`<c-x><c-n>`	当前缓冲区中的关键字	`:h i^x^n`
`<c-x><c-k>`	字典（请参阅 `:h 'dictionary'`）中的关键字	`:h i^x^k`
`<c-x><c-t>`	同义词字典（请参阅 `:h 'thesaurus'`）中的关键字	`:h i^x^t`
`<c-x><c-i>`	当前文件以及包含的文件中的关键字	`:h i^x^i`
`<c-x><c-]>`	标签	`:h i^x^]`
`<c-x><c-f>`	文件名	`:h i^x^f`
`<c-x><c-d>`	定义或宏定义	`:h i^x^d`
`<c-x><c-v>`	Vim 命令	`:h i^x^v`
`<c-x><c-u>`	用户自定义补全（通过 `'completefunc'` 定义）	`:h i^x^u`
`<c-x><c-o>`	Omni Completion（通过 `'omnifunc'` 定义）	`:h i^x^o`
`<c-x>s`	拼写建议	`:h i^Xs`

尽管用户自定义补全与 Omni Completion 是不同的，但他们做的事情基本一致。共同点在于，他们都是一个监听当前光标位置的函数，返回值为一系列的补全建议。用户自定义补全是由用户定义的，基于用户的个人用途，因此你可以根据自己的喜好和需求随意定制。而 Omni Completion 是针对文件类型的补全，比如在 C 语言中补全一个结构体（struct）的成员（members），或者补全一个类的方法，因而它通常都是由文件类型插件设置和调用的。

如果你设置了 `'complete'` 选项，那么你就可以在一次操作中采用多种补全方案。这个选项默认包含了多种可能性，因此请按照自己的需求来配置。你可以通过 `<c-n>` 来调用下一个补全建议，或通过 `<c-p>` 来调用上一个补全建议。当然，这两个映射同样可以直接调用补全函数。请参阅 `:h i^n` 与 `:h 'complete'` 来获得更多帮助。

如果你想配置弹出菜单的行为，请一定要看一看 `:h 'completeopt'` 这篇帮助文档。默认的配置已经不错了，但我个人（原作者）更倾向于把 "noselect" 加上。

请参阅以下文档获取更多帮助：

```
:h ins-completion
:h popupmenu-keys
:h new-omni-completion
```

返回主目录 ⤴

动作，操作符，文本对象

动作也就是指移动光标的操作，你肯定很熟悉 `h`、`j`、`k` 和 `l`，以及 `w` 和 `b`。但其实，`/` 也是一个动作。他们都可以搭配数字使用，比如 `2?the<cr>` 可以将光标移动到倒数第二个 "the" 出现的位置。

以下会列出一些常用的动作。你也可以通过 `:h navigation` 来获取更多的帮助。

操作符是对某个区域文本执行的操作。比如，`d`、`~`、`gU` 和 `>` 都是操作符。这些操作符既可以在普通模式下使用，也可以在可视模式下使用。在普通模式中，顺序是先按操作符，再按动作指令，比如 `>j`。在可视模式中，选中区域后直接按操作符就可以，比如 `vjd`。

与动作一样，操作符也可以搭配数字使用，比如 `2gUw` 可以将当前单词以及下一个单词转成大写。由于动作和操作符都可以搭配数字使用，因此 `2gU2w` 与执行两次 `gU2w` 效果是相同的。

请参阅 `:h operator` 来查看所有的操作符。你也可以通过 `:set tildeop` 命令把 `~` 也变成一个操作符

值得注意的是，动作是单向的，而**文本对象**是双向的。文本对象不仅作用于符号（比如括号、中括号和大括号等）标记的范围内，也作用于整个单词、整个句子等其他情况。

文本对象不能用于普通模式中移动光标的操作，因为光标还没有智能到可以向两个方向同时跳转。但这个功能可以在可视模式中实现，因为在对象的一端选中的情况下，光标只需要跳转到另一端就可以了。

文本对象操作一般用 `i` 或 `a` 加上对象标识符操作，其中 `i` 表示在对象内（英文 inner）操作，`a` 表示对整个对象（英文 around）操作，这时开头和结尾的空格都会被考虑进来。举个例子，`diw` 可以删除当前单词，`ci(` 可以改变括号中的内容。

文本对象同样可以与数字搭配使用。比如，像 `((( )))` 这样的文本，假如光标位于最内层的括号上或最内层的括号内，那么 `d2a(` 将会删除从最内层开始的两对括号，以及他们之间的所有内容。其实，`d2a(` 这个操作等同于 `2da(`。在 Vim 的命令中，如果有两处都可以接收数字作为参数，那么最终结果就等同于两个数字相乘。在这里，`d` 与 `a(` 都是可以接收参数的，一个参数是 1，另一个是 2，我们可以把它们相乘然后放到最前面。

请参阅 `:h text-objects` 来获取更多关于文本对象的帮助。

返回主目录 ⤴

自动命令

在特定的情况下，Vim 会传出事件。如果你想针对这些事件执行回调方法，那么就需要用到自动命令这个功能。

如果没有了自动命令，那你基本上是用不了 Vim 的。自动命令一直都在执行，只是很多时候你没有注意到。不信的话，可以执行命令 `:au` ，不要被结果吓到，这些是当前有效的所有自动命令。

请使用 `:h {event}` 来查看 Vim 中所有事件的列表，你也可以参考 `:h autocmd-events-abc` 来获取关于事件的更多帮助。

一个很常用的例子，就是针对文件类型执行某些设置：

```
autocmd FileType ruby setlocal shiftwidth=2
softtabstop=2 comments-=:#
```

但是缓冲区是如何知道当前的文件中包含 Ruby 代码呢？这其实是另一个自动命令检测的到的，然后把文件类型设置成为 Ruby，这样就触发了上面的 `FileType` 事件。

在配置 vimrc 的时候，一般第一行加进去的就是 `filetype on`。这就意味着，Vim 启动时会读取 `filetype.vim` 文件，然后根据文件类型来触发相应的自动命令。

如果你勇于尝试，可以查看下 `:e $VIMRUNTIME/filetype.vim`，然后在输出中搜索 "Ruby"。这样，你就会发现其实 Vim 只是通过文件扩展名 `.rb` 判断某个文件是不是 Ruby 的。

注意：对于相同事件，如果有多个自动命令，那么自动命令会按照定义时的顺序执行。通过 `:au` 就可以查看它们的执行顺序。

```
au BufNewFile,BufRead *.rb,*.rbw setf ruby
```

`BufNewFile` 与 `BufRead` 事件是被写在 Vim 源文件中的。因此，每当你通过 `:e` 或者类似的命令打开文件，这两个事件都会触发。然后，就是读取 `filetype.vim` 文件来判断打开的文件类型。

简单来说，事件和自动命令在 Vim 中的应用十分广泛。而且，Vim 为我们留出了一些易用的接口，方便用户配置适合自己的事件驱动回调。

请参阅 `:h autocommand` 来获取更多帮助

返回主目录 ⤴

变更历史，跳转历史

在 Vim 中，用户最近 100 次的文字改动都会被保存在**变更历史**中。如果在同一行有多个小改动，那么 Vim 会把它们合并成一个。尽管内容改动会合并，但作用的位置还是会只记录下最后一次改动的位置。

在你移动光标或跳转的时候，每一次的移动或跳转前的位置会被记录到**跳转历史**中。类似地，跳转历史也可以最多保存 100 条记录。对于每个窗口，跳转记录是独立的。但当你分离窗口时（比如使用 `:split` 命令），跳转历史会被复制过去。

Vim 中的跳转命令，包括 `'`、`` ` ``、`G`、`/`、`?`、`n`、`N`、`%`、`(`、`)`、`[[`、`]]`、`{`、`}`、`:s`、`:tag`、`L`、`M`、`H` 以及开始编辑一个新文件的命令。

列表	显示所有条目	跳转到上一个位置	跳转到下一个位置
跳转历史	`:jumps`	`[count]<c-o>`	`[count]<c-i>`
变更历史	`:changes`	`[count]g;`	`[count]g,`

如果你执行第二列的命令显示所有条目，这时 Vim 会用 `>` 标记来为你指示当前位置。通常这个标记位于 1 的下方，也就代表最后一次的位置。

如果你希望关闭 Vim 之后还保留这些条目，请参阅 `:h viminfo-'` 来获取更多帮助。

注意：上面提到过，最后一次跳转前的位置也会记录在标注中，也可以通过连按 `\\` 或 `''` 跳转到那个位置

请参阅以下两个命令来获取更多帮助：

```
:h changelist
:h jumplist
```

返回主目录 ⤴

内容变更历史记录

Vim 会记录文本改变之前的状态。因此，你可以使用「撤销」操作 `u` 来取消更改，也可以通过「重做」操作 `Ctrl + r` 来恢复更改。

值得注意的是，Vim 采用 tree 数据结构来存储内容变更的历史记录，而不是采用 queue。你的每次改动都会成为存储为树的节点。而且，除了第一次改动（根节点），之后的每次改动都可以找到一个对应的父节点。每一个节点都会记录改动的内容和时间。其中，「分支」代表从任一节点到根

节点的路径。当你进行了撤销操作，然后又输入了新的内容，这时候就相当于创建了分支。这个原理和 git 中的 branch（分支）十分类似。

考虑以下这一系列按键操作：

```
ifoo<esc>
obar<esc>
obaz<esc>
u
oquux<exc>
```

那么现在，Vim 中会显示三行文本，分别是 "foo"、"bar" 和 "quux"。这时候，存储的树形结构如下：

```
    foo(1)
      /
   bar(2)
  /      \
baz(3)   quux(4)
```

这个树形结构共包含四次改动，括号中的数字就代表时间顺序。

现在，我们有两种方式遍历这个树结构。一种叫「按分支遍历」，一种叫「按时间遍历」。

撤销 `u` 与重做 `Ctrl + r` 操作是按分支遍历。对于上面的例子，现在我们有三行字符。这时候按 `u` 会回退到 "bar" 节点，如果再按一次 `u` 则会回退到 "foo" 节点。这时，如果我们按下 `Ctrl + r` 就会前进至 "bar" 节点，再按一次就回前进至 "quux" 节点。在这种方式下，我们无法访问到兄弟节点（即 "baz" 节点）。

与之对应的是按时间遍历，对应的按键是 `g-` 和 `g+`。对于上面的例子，按下 `g-` 会首先回退到 "baz" 节点。再次按下 `g-` 会回退到 "bar" 节点。

命令/按键	执行效果
`[count]u` 或 `:undo [count]`	回退到 `[count]` 次改动之前
`[count] <c-r>` 或 `:redo [count]`	重做 `[count]` 次改动
`U`	回退至最新的改动
`[count]g-` 或 `:earlier [count]?`	根据时间回退到 `[count]` 次改动之前。"?" 为 "s"、"m"、"h"、"d" 或 "f"之一。例如，`:earlier 2d` 会回退到两天之前。`:earlier 1f` 则会回退到最近一次文件保存时的内容
`[count]g+` 或 `:later [count]?`	类似 `g-`，但方向相反

内容变更记录会储存在内存中，当 Vim 退出时就会清空。如果需要持久化存储内容变更记录，请参阅备份文件，交换文件，撤销文件以及 viminfo 文件的处理章节的内容。

如果你觉得这一部分的内容难以理解，请参阅 undotree，这是一个可视化管理内容变更历史记录的插件。类似的还有 vim-mundo。

请参阅以下链接获取更多帮助：

```
:h undo.txt
:h usr_32
```

返回主目录 ⤴

全局位置信息表，局部位置信息表

在某一个动作返回一系列「位置」的时候，我们可以利用「全局位置信息表」和「局部位置信息表」来存储这些位置信息，方便以后跳转回对应的位置。每一个存储的位置包括文件名、行号和列号。

比如，编译代码是出现错误，这时候我们就可以把错误的位置直接显示在全局位置信息表，或者通过外部抓取工具使位置显示在局部位置信息表中。

尽管我们也可以把这些信息显示到一个空格缓冲区中，但用这两个信息表显示的好处在于接口调用很方便，而且也便于浏览输出。

Vim 中，全局位置信息表只能有一个，但每一个窗口都可以有自己的局部位置信息表。这两个信息表的外观看上去很类似，但在操作上会稍有不同。

以下为两者的操作比较：

动作	全局位置信息表	局部位置信息表
打开窗口	`:copen`	`:lopen`
关闭窗口	`:cclose`	`:lclose`
下一个条目	`:cnext`	`:lnext`
上一个条目	`:cprevious`	`:lprevious`
第一个条目	`:cfirst`	`:lfirst`
最后一个条目	`:clast`	`:llast`

请参阅 `:h :cc` 以及底下的内容，来获取更多命令的帮助。

应用实例：
如果我们想用 `grep` 递归地在当前文件夹中寻找某个关键词，然后把输出结果放到全局位置信息表中，只需要这样：

```
:let &grepprg = 'grep -Rn $* .'
:grep! foo
<grep output - hit enter>
:copen
```

执行了上面的代码，你就能看到所有包含字符串 "foo" 的文件名以及匹配到的相关字段都会显示在全局位置信息表中。

返回主目录 ⤴

宏

你可以在 Vim 中录制一系列按键，并把他们存储到寄存器中。对于一些需要临时使用多次的一系列操作，把它们作为宏保存起来会显著地提升效率。对于一些复杂的操作，建议使用 Vim 脚本来实现。

- 首先，按下 `q`，然后按下你想要保存的寄存器，任何小写字母都可以。比如我们来把它保存到 `q` 这个寄存器中。按下 `qq`，你会发现命令行里已经显示了 "recording @q"。
- 如果你已经录制完成，那么只需要再按一次 `q` 就可以结束录制。
- 如果你想调用刚才录制的宏，只需要 `[count]@q`
- 如果你想调用上一次使用的宏，只需要 `[count]@@`

实例 1:

一个插入字符串 "abc" 后换行的宏，重复调用十次：

```
qq
iabc<cr><esc>
q
10@q
```

（对于上面这个功能，你同样可以通过如下的按键： `o` `a` `b` `c` 然后 `ESC` 然后 `1` `0` `.` 来实现）。

实例 2:

一个在每行前都加上行号的宏。从第一行开始，行号为 1，后面依次递增。我们可以通过 `Ctrl` + `a` 来实现递增的行号，在定义宏的时候，它会显示成 `^A`。

```
qq
0yf jP0^A
q
1000 @q
```

这里能实现功能，是因为我们假定了文件最多只有 1000 行。但更好的方式是使用「递归」宏，它会一直执行，知道不能执行为止：

```
qq
0yf jP0^A@q
q
@q
```

（对于上面这个插入行号的功能，如果你不愿意使用宏，同样可以通过这段按键操作来实现：`:%s/^/\=line('.') . '. '`）。

这里向大家展示了如何不用宏来达到相应的效果，但要注意，这些不用宏的实现方式只适用于这些简单的示例。对于一些比较复杂的自动化操作，你确实应该考虑使用宏。

请参阅以下文档获取更多帮助：

```
:h recording
:h 'lazyredraw'
```

返回主目录 ⤴

颜色主题

颜色主题可以把你的 Vim 变得更漂亮。Vim 是由多个组件构成的，我们可以给每一个组件都设置不同的文字颜色、背景颜色以及文字加粗等等。比如，我们可以通过这个命令来设置背景颜色：

```
:highlight Normal ctermbg=1 guibg=red
```

执行后你会发现，现在背景颜色变成红色了。请参阅 `:h :highlight` 来获取更多帮助。

其实，颜色主题就是一系列的 `:highlight` 命令的集合。

事实上，大部分颜色主题都包含两套配置。一套适用于例如 xterm 和 iTerm 这样的终端环境（使用前缀 `cterm`），另一套适用于例如 gvim 和 MacVim 的图形界面环境（使用前缀 `gui`）。对于上面的例子，`ctermbg` 就是针对终端环境的，而 `guibg` 就是针对图形界面环境的。

如果你下载了一个颜色主题，并且在终端环境中打开了 Vim，然后发现显示的颜色与主题截图中差别很大，那很可能是配置文件只设置了图形界面环境的颜色。反之同理，如果你使用的是图形界面环境，发现显示颜色有问题，那就很可能是配置文件只设置了终端环境的颜色。

第二种情况（图形界面环境的显示问题）其实不难解决。如果你使用的是 Neovim 或者 Vim 7.4.1830 的后续版本，可以通过打开真彩色设置来解决显示问题。这就可以让终端环境的 Vim 使用 GUI 的颜色定义，但首先，你要确认一下你的终端环境和环境内的组件（比如 tmux）是否都支持真彩色。可以看一下这篇文档，描述的十分详细。

请参阅以下文档或链接来获取更多帮助：

- `:h 'termguicolors'`
- 主题列表
- 自定义主题中的颜色

返回主目录 ⤴

折叠

每一部分文字（或者代码）都会有特定的结构。对于存在结构的文字和代码，也就意味着它们可以按照一定的逻辑分割成不同区域。Vim 中的折叠功能，就是按照特定的逻辑把文字和代码折叠成一行，并显示一些简短的描述。折叠功能涉及到很多操作，而且折叠功能可以嵌套使用。

在 Vim 中，有以下 6 中折叠类型：

折叠方式	概述
diff	在「比较窗口」中折叠未改变的文本
expr	使用 `'foldexpr'` 来创建新的折叠逻辑
indent	基于缩进折叠
manual	使用 `zf`、`zF` 或 `:fold` 来自定义折叠
marker	根据特定的文本标记折叠（通常用于代码注释）
syntax	根据语法折叠，比如折叠 `if` 代码块

注意：折叠功能可能会显著地影响性能。如果你在使用折叠功能的时候出现了打字卡顿之类的问题，请考虑使用 FastFold 插件。这个插件可以让 Vim 按需更新折叠内容，而不是一直调用。

请参阅以下文档获取更多帮助：

```
:h usr_28
:h folds
```

会话

如果你保存了当前的「视图」（请参阅 `:h :mkview`），那么当前窗口、配置和按键映射都会被保存下来（请参阅 `:h :loadview`）。

「会话」就是存储所有窗口的相关设置，以及全局设置。简单来说，就是给当前的 Vim 运行实例拍个照，然后把相关信息存储到会话文件中。存储之后的改动就不会在会话文件中显示，你只需要在改动后更新一下会话文件就可以了。

你可以把当前工作的「项目」存储起来，然后可以在不同的「项目」之间切换。

现在就来试试吧。打开几个窗口和标签，然后执行 `:mksession Foo.vim`。如果你没有指定文件名，那就会默认保存为 `Session.vim`。这个文件会保存在当前的目录下，你可以通过 `:pwd` 来显示当前路径。重启 Vim 之后，你只需要执行 `:source Foo.vim`，就可以恢复刚才的会话了。所有的缓冲区、窗口布局、按键映射以及工作路径都会恢复到保存时的状态。

其实 Vim 的会话文件就只是 Vim 命令的集合。你可以通过命令 `:vs Foo.vim` 来看看会话文件中究竟有什么。

你可以决定 Vim 会话中究竟要保存哪些配置，只需要设置一下 `'sessionoptions'` 就可以了。

为了方便开发，Vim 把最后一次调用或写入的会话赋值给了一个内部变量 `v:this_session`。

请参阅以下文档来获取更多帮助：

```
:h Session
:h 'sessionoptions'
:h v:this_session
```

局部化

以上提到的很多概念，都有一个局部化（非全局）的版本：

全局	局部	作用域	帮助文档
`:set`	`:setlocal`	缓冲区或窗口	`:h local-options`
`:map`	`:map <buffer>`	缓冲区	`:h :map-local`
`:autocmd`	`:autocmd * <buffer>`	缓冲区	`:h autocmd-buflocal`
`:cd`	`:lcd`	窗口	`:h :lcd`
`:<leader>`	`:<localleader>`	缓冲区	`:h maploacalleader`

变量也有不同的作用域，详细内容请参考 Vim scripting 的文档。

用法

获取离线帮助

Vim 自带了一套很完善的帮助文档，它们是一个个有固定排版格式的文本文件，通过标签可以访问这些文件的特定位置。

在开始之前先读一下这个章节：`:help :help`。执行这个命令以后会在新窗口打开 `$VIMRUNTIME/doc/helphelp.txt` 文件并跳转到这个文件中 `:help` 标签的位置。

一些关于帮助主题的简单规则：

- 用单引号把文本包起来表示选项，如：`:h 'textwidth'`
- 以小括号结尾表示 VimL 函数，如：`:h reverse()`
- 以英文冒号开头表示命令，如：`:h :echo`

使用快捷键 `<c-d>`（这是 `ctrl`+`d`）来列出所有包含你当前输入的内容的帮助主题。如：`:h tab<c-d>` 会列出所有包含 `tab` 主题，从 `softtabstop` 到 `setting-guitablabel`（译者注：根据安装的插件不同列出的选项也会不同）。

你想查看所有的 VimL 方法吗？很简单，只要输入：`:h ()<c-d>` 就可以了。你想查看所有与窗口相关的函数吗？输入 `:h win*()<c-d>`。

相信你很快就能掌握这些技巧，但是在刚开始的时候，你可能对于该通过什么进行查找一点线索都没有。这时你可以想象一些与要查找的内容相关的关键字，再让 `:helpgrep` 来帮忙。

```
:helpgrep backwards
```

上面的命令会在所有的帮助文件中搜索“backwards”，然后跳转到第一个匹配的位置。所有的匹配位置都会被添加到全局位置信息表，用 `:cp / :cn` 可以在匹配位置之间进行切换。或者用 `:copen` 命令来打开全局位置信息表，将光标定位到你想要的位置，再按 回车就可以跳转到该匹配项。详细说明请参考 `:h quickfix`。

获取离线帮助（补充）

这个列表最初发表在 vim_dev，由 @chrisbra 编辑的，他是 Vim 开发人员中最活跃的一个。

经过一些微小的改动后，重新发布到了这里。

如果你知道你想要找什么，使用帮助系统的搜索会更简单一些，因为搜索出的主题都带有固定的格式。

而且帮助系统中的主题包含了你当前使用的 Vim 版本的所特有特性，而网上那些已经过时或者是早期发布的话题是不会包含这些的。

因此学习使用帮助系统以及它所用的语言是很有必要的。这里是一些例子（不一定全，我有可能忘了一些什么）。

（译者注：下面列表中提及的都是如何指定搜索主题以便快速准确的找到你想要的帮助）

1. 选项要用单引号引起来。用 `:h 'list'` 来查看列表选项帮助。只有你明确的知道你要找这么一个选项的时候才可以这么做，不然的话你可以用 `:h options.txt` 来打开所有选项的帮助页面，再用正则表达式进行搜索，如：`/width`。某些选项有它们自己的命名空间，如：`:h cpo-a`，`:h cpo-A`，`:h cpo-b` 等等。

2. 普通模式的命令不能用冒号作为前缀。使用 `:h gt` 来转到“gt”命令的帮助页面。
3. 正则表达式以“/”开头，所以 `:h /\+` 会带你到正则表达式中量词“+”的帮助页面。
4. 组合键经常以一个字母开头表示它们可以在哪些模式中使用。如：`:h i_CTRL-X` 会带你到插入模式下的 CTRL-X 命令的用法帮助页面，这是一个自动完成类的组合键。需要注意的是某些键是有固定写法的，如 Control 键写成 CTRL。还有，查找普通模式下的组合键帮助时，可以省略开头的字母“n”，如：`:h CTRL-A`。而 `:h c_CTRL-A`（译者注：原文为 `:h c_CRTL-R`，感觉改为 A 更符合上下文语境）会解释 CTRL-A 在命令模式下输入命令时的作用；`:h v_CTRL-A` 说的是在可见模式下把光标所在处的数字加 1；`:h g_CTRL-A` 则说的是 g 命令（你需要先按 "g" 的命令）。这里的 "g" 代表一个普通的命令，这个命令总是与其它的按键组合使用才生效，与 "z" 开始的命令相似。
5. 寄存器是以 "quote" 开头的。如：`:h quote:` （译者注：原文为`:h quote`，感觉作者想以":"来举例）来查看关于":"寄存器的说明。
6. 关于 Vim 脚本（VimL）的帮助都在 `:h eval.txt` 里。而某些方面的语言可以使用 `:h expr-X` 获取帮助，其中的 'X' 是一个特定的字符，如：`:h expr-!` 会跳转到描述 VimL 中'!'（非）的章节。另外一个重要提示，可以使用 `:h function-list` 来查看所有函数的简要描述，列表中包括函数名和一句话描述。
7. 关于映射都可以在 `:h map.txt` 中找到。通过 `:h mapmode-i` 来查找 `:imap` 命令的相关信息；通过 `:h map-topic` 来查找专门针对映射的帮助（译者注：topic 为一个占位符，正如上面的字符 'X' 一样，在实际使用中需要替换成相应的单词）（如：`:h :map-local` 查询本地 buffer 的映射，`:h map-bar` 查询如何在映射中处理'|'）。
8. 命令定义用 "command-" 开头，如用 `:h command-bar` 来查看自定义命令中'!'的作用。
9. 窗口管理类的命令是以 "CTRL-W" 开头的，所以你可以用 `:h CTRL-W_*` 来查找相应的帮助（译者注：'*'同样为占位符）（如：`:h CTRL-W_p` 查看切换到之前访问的窗口命令的解释）。如果你想找窗口处理的命令，还可以通过访问 `:h windows.txt` 并逐行向下浏览，所有窗口管理的命令都在这里了。
10. 执行类的命令以":"开头，即：`:h :s` 讲的是 ":s" 命令。
11. 在输入某个话题时按 CTRL-D，让 Vim 列出所有的近似项辅助你输入。

12. 用 `:helpgrep` 在所有的帮助页面（通常还包括了已安装的插件的帮助页面）中进行搜索。参考 `:h :helpgrep` 来了解如何使用。当你搜索了一个话题之后，所有的匹配结果都被保存到了全局位置信息表（或局部位置信息表）当中，可以通过 `:copen` 或 `:lopen` 打开。在打开的窗口中可能通过 `/` 对搜索结果进行进一步的过滤。
13. `:h helphelp` 里介绍了如何使用帮助系统。
14. 用户手册。它采用了一种对初学者更加友好的方式来展示帮助话题。用 `:h usr_toc.txt` 打开目录（你可能已经猜到这个命令的用处了）。浏览用户手册能帮助你找出某些你想了解的话题，如你可以在第 24 章看到关于“复合字符”以及“输入特殊字符”的讲解（用 `:h usr_24.txt` 可以快速打开相关章节）。
15. 高亮分组的帮助以 `hl-` 开头。如：`:h hl-WarningMsg` 说的是警告信息分组的高亮。
16. 语法高亮以 `:syc-` 开头，如：`:h :syn-conceal` 讲的是 `:syn` 命令的对于隐藏字符是如何显示的。
17. 快速修复命令以 `:c` 开头，而位置列表命令以 `:l` 开头。
18. `:h BufWinLeave` 讲的是 BufWinLeave 自动命令。还有，`:h autocommand-events`（译者注：原文是 `:h autocommands-events`，但是没有该帮助）讲的是所有可用的事件。
19. 启动参数都以“-”开头，如：`:h -f` 会告诉你 Vim 中 “-f” 参数的作用。
20. 额外的特性都以“+”开头，如：`:h +conceal` 讲的是关于隐藏字符的支持。
21. 错误代码可以在帮助系统中直接查到。`:h E297` 会带你到关于这一错误的详细解释。但是有时并没有转到错误描述，而是列出了经常导出这一错误的 Vim 命令，如 `:h E128`（译者注：原文为 `:h hE128`，但是并没有该帮助）会直接跳转到 `:function` 命令。
22. 关于包含的语法文件的文档的帮助话题格式是 `:h ft-*-syntax`。如：`:h ft-c-syntax` 说的就是 C 语言语法文件以及它所提供的选项。有的语法文件还会带有自动完成（`:h ft-php-omni`）或文件类型插件（`:h ft-tex-plugin`）相关的章节可以查看。

另外在每个帮助页的顶端通常会包含一个用户文档链接（更多的从从用户的角度出发来主角命令的功能和用法，不涉及那么多细节）。如：`:h pattern.txt` 里包含了 `:h 03.9` 和 `:h usr_27` 两个章节的链接。

获取在线帮助

如果你遇到了无法解决的问题，或者需要指引的话，可以参考 Vim 使用邮件列表。 IRC 也是一个很不错的资源。 Freenode 上的 `#vim` 频道很庞大，并且里面有许多乐于助人的人。

如果你想给 Vim 提交 Bug 的话，可以使用 vim_dev 邮件列表。

执行自动命令

你可以触发任何事件，如：`:doautocmd BufRead`。

用户自定义事件

对于插件而言，创建你自己的自定义事件有时非常有用。

```
function! Chibby()
    " A lot of stuff is happening here.
    " And at last..
    doautocmd User ChibbyExit
endfunction
```

现在你插件的用户可以在 Chibby 执行完成之后做任何他想做的事情：

```
autocmd User ChibbyExit call ChibbyCleanup()
```

顺便提一句，如果在使用 `:autocmd` 或 `:doautocmd` 时没有捕捉异常，那么会输出 "No matching autocommands" 信息。这也是为什么许多插件用 `silent doautocmd ...` 的原因。但是这也会有不足，那就是你不能再在 :autocmd 中使用 `echo "foo"` 了，取而代之的是你要使用 `unsilent echo "foo"` 来输出。

这就是为什么要在触发事件之前先判断事件是否存在的原因，

```
if exists('#User#ChibbyExit')
  doautocmd User ChibbyExit
endif
```

帮助文档：`:h User`

事件嵌套

默认情况下，自动命令不能嵌套！如果某个自动命令执行了一个命令，这个命令再依次触发其它的事件，这是不可能的。

例如你想在每次启动 Vim 的时候自动打开你的 vimrc 文件：

```
autocmd VimEnter * edit $MYVIMRC
```

当你启动 Vim 的时候，它会帮你打开你的 vimrc 文件，但是你很快会注意到这个文件没有任何的高亮，尽管平时它是正常可以高亮的。

问题在于你的非嵌套自动命令 `:edit` 不会触发“BufRead”事件，所以并不会把文件类型设置成“vim”，进而 `$VIMRUNTIME/syntax/vim.vim` 永远不会被引入。详细信息请参考：`:au BufRead *.vim`。要想完成上面所说的需求，使用下面这个命令：

```
autocmd VimEnter * nested edit $MYVIMRC
```

帮助文档：`:h autocmd-nested`

剪切板

如果你想在没有 GUI 支持的 Unix 系统中使用 Vim 的 `'clipboard'` 选项，则需要 `+clipboard` 以及可选的 `+xterm_clipboard` 两个特性支持。

帮助文档：

```
:h 'clipboard'
:h gui-clipboard
:h gui-selections
```

另外请参考：持续粘贴（为什么我每次都要设置 'paste' 模式

剪贴板的使用（Windows, OSX）

Windows 自带了剪贴板，OSX 则带了一个粘贴板

在这两个系统中都可以用大家习惯用的 `ctrl+c / cmd+c` 复制选择的文本，然后在另外一个应用中用 `ctrl+v / cmd+v` 进行粘贴。

需要注意的是复制的文本已经被发送到了剪贴板，所以你在粘贴复制的内容之前关闭这个应用是没有任何问题的。

每次复制的时候，都会向剪贴板寄存器 `*` 中写入数据。 而在 Vim 中分别使用 `"*y` 和 `"*p` 来进行复制（yank) 和 粘贴（paste)。

如果你不想每次操作都要指定 `*` 寄存器，可以在你的 vimrc 中添加如下配置：

```
set clipboard=unnamed
```

通常情况下复制/删除/放入操作会往 `"` 寄存器中写入数据，而加上了上面的配置之后 `*` 寄存器也会被写入同样数据，因此简单的使用 `y` 和 `p` 就可以复制粘贴了。

我再说一遍：使用上面的选项意味着每一次的复制/粘贴，即使在同一个 Vim 窗口里，都会修改剪贴板的内容。你自己决定上面的选项是否适合。

如果你觉得输入 `y` 还是太麻烦的话，可以使用下面的设置把在可视模式下选择的内容发送到剪贴板：

```
set clipboard=unnamed,autoselect
set guioptions+=a
```

帮助文档：

```
:h clipboard-unnamed
:h autoselect
:h 'go_a'
```

剪贴板的使用（Linux, BSD, ...）

如果你的系统使用了 X 图形界面，事情会变得有一点不同。X 图形界面实现了 X 窗口系统协议, 这个协议在 1987 年发布的主版本 11，因此 X 也通常被称为 X11。

在 X10 版本中，剪贴缓冲区被用来实现像 *clipboard* 一样由 X 来复制文本，并且可以被所有的程序访问。现在这个机制在 X 中还存在，但是已经过时了，很多程序都不再使用这一机制。

近年来数据在程序之间是通过选择进行传递的。一共有三种选择，经常用到的有两种：PRIMARY 和 CLIPBOARD。

选择的工作工模大致是这样的：

```
Program A: <ctrl+c>
Program A: 声称对 CLIPBOARD 的所有权
Program B: <ctrl+v>
Program B: 发现CLIPBOARD的所有权被Program A持有
Program B: 从Program A请求数据
Program A: 响应这个请求并发送数据给Program B
Program B: 从Program A接收数据并插入到窗口中
```

选择	何时使用	如何粘贴	如何在 Vim 中访问
PRIMARY	选择文本	鼠标中键, shift+insert	`*` 寄存器
CLIPBOARD	选择文本并按 `ctrl+c`	`ctrl+v`	`+`寄存器

注意：X 服务器并不会保存选择（不仅仅是 CLIPBOARD 选择）！因此在关闭了相应的程序后，你用 `ctrl+c` 复制的内容将丢失。

使用 `"*p` 来贴粘 PRIMARY 选择中的内容，或者使用 `"+y1G` 来将整个文件的内容复制到 CLIPBOARD 选择。

如果你需要经常访问这两个寄存器，可以考虑使用如下配置：

```
set clipboard^=unnamed          " * 寄存器
" 或者
set clipboard^=unnamedplus      " + 寄存器
```

（`^=` 用来将设置的值加到默认值之前，详见：`:h :set^=`）

这会使得所有复制/删除/放入操作使用 `*` 或 `+` 寄存器代替默认的未命令寄存器 `"`。之后你就可以直接使用 `y` 或 `p` 访问你的 X 选择了。

帮助文档：

```
:h clipboard-unnamed
:h clipboard-unnamedplus
```

打开文件时恢复光标位置

如果没有这个设置，每次打开文件时光标都将定位在第一行。而加入了这个设置以后，你就可以恢复到上次关闭文件时光标所在的位置了。

将下面的配置添加到你的 vimrc 文件：

```
autocmd BufReadPost *
    \ if line("'\"") > 1 && line("'\"") <= line("$") |
    \   exe "normal! g`\"" |
    \ endif
```

这是通过判断之前的光标位置是否存在（文件可能被其它程序修改而导致所记录的位置已经不存在了），如果存在的话就执行 `g`"` （转到你离开时的光标位置但是不更改跳转列表）。

这需要使用 viminfo 文件：`:h viminfo-`。

临时文件

根据选项的不同， Vim 最多会创建 4 种工作文件。

备份文件

你可以让 Vim 在将修改写入到文件之前先备份原文件。默认情况下， Vim 会保存一个备份文件但是当修改成功写入后会立即删除它（`:set writebackup`）。如果你想一直保留这个备份文件的话，可以使用 `:set backup`。而如果你想禁用备份功能的话，可以使用 `:set nobackup nowritebackup`。

咱们来看一下上次我在 vimrc 中改了什么：

```
$ diff ~/.vim/vimrc ~/.vim/files/backup/vimrc-vimbackup
390d389
< command! -bar -nargs=* -complete=help H helpgrep
<args>
```

帮助文档：`:h backup`

交换文件

假设你有一个非常棒的科幻小说的构思。在按照故事情节已经写了好几个小时几十万字的时候..忽然停电了！而那时你才想起来你上次保存 `~/来自外太空的邪恶入侵者.txt` 是在.. 好吧，你从来没有保存过。

但是并非没有希望了！在编辑某个文件的时候， Vim 会创建一个交换文件，里面保存的是对当前文件所有未保存的修改。自己试一下，打开任意的文件，并使用 `:swapname` 获得当前的交换文件的保存路径。你也可以将 `:set noswapfile` 加入到 vimrc 中来禁用交换文件。

默认情况下，交换文件会自动保存在被编辑文件所在的目录下，文件名以 `.file.swp` 后缀结尾，每当你修改了超过 200 个字符或是在之前 4 秒内没有任何动作时更新它的内容，在你不再编辑这个文件的时候会被删除。你可以自己修改这些数字，详见：`:h 'updatecount'` 和 `:h 'updatetime'`。

而在断电时，交换文件并不会被删除。当你再次打开 `vim ~/来自外太空的邪恶入侵者.txt` 时， Vim 会提示你恢复这个文件。

帮助文档：`:h swap-file` 和 `:h usr_11`

撤销文件

内容变更历史记录是保存在内存中的，并且会在 Vim 退出时清空。如果你想让它持久化到磁盘中，可以设置 `:set undofile`。这会把文件 `~/foo.c` 的撤销文件保存在 `~/foo.c.un~`。

帮助文档：`:h 'undofile'` 和 `:h undo-persistence`

viminfo 文件

备份文件、交换文件和撤销文件都是与文本状态相关的，而 viminfo 文件是用来保存在 Vim 退出时可能会丢失的其它的信息的。包括历史记录（命令历史、搜索历史、输入历史）、寄存器内容、标注、缓冲区列表、全局变量等等。

默认情况下，viminfo 被保存在 `~/.viminfo`。

帮助文档：`:h viminfo` 和 `:h 'viminfo'`

临时文件管理设置示例

如果你跟我一样，也喜欢把这些文件放到一个位置（如：`~/.vim/files`）的话，可以使用下面的配置：

```
" 如果文件夹不存在，则新建文件夹
if !isdirectory($HOME.'/.vim/files') &&
exists('*mkdir')
  call mkdir($HOME.'/.vim/files')
endif

" 备份文件
set backup
set backupdir   =$HOME/.vim/files/backup/
set backupext   =-vimbackup
set backupskip  =
" 交换文件
set directory   =$HOME/.vim/files/swap//
set updatecount =100
" 撤销文件
set undofile
set undodir     =$HOME/.vim/files/undo/
" viminfo 文件
set viminfo     ='100,n$HOME/.vim/files/info/viminfo
```

注意：如果你在一个多用户系统中编辑某个文件时， Vim 提示你交换文件已经存在的话，可能是因为有其他的用户此时正在编辑这个文件。而如果将交换文件放到自己的 home 目录的话，这个功能就失效了。因此服务器非常不建议将这些文件修改到 HOME 目录，避免多人同时编辑一个文件，却没有任何警告。

编辑远程文件

Vim 自带的 netrw 插件支持对远程文件的编辑。实际上它将远程的文件通过 scp 复制到本地的临时文件中，再用那个文件打开一个缓冲区，然后在保存时把文件再复制回远程位置。

下面的命令在你本地的 VIM 配置与 SSH 远程服务器上管理员想让你使用的配置有冲突时尤其有用：

```
:e scp://bram@awesome.site.com/.vimrc
```

如果你已经设置了 `~/.ssh/config`，SSH 会自动读取这里的配置：

```
Host awesome
    HostName awesome.site.com
    Port 1234
    User bram
```

如果你的 `~/.ssh/config` 中有以上的内容，那么下面的命令就可以正常执行了：

```
:e scp://awesome/.vimrc
```

可以用同样的方法编辑 `~/.netrc`，详见：`:h netrc-netrc`。

确保你已经看过了 `:h netrw-ssh-hack` 和 `:h g:netrw_ssh_cmd`。

另外一种编辑远程文件的方法是使用 sshfs，它会用 FUSE 来挂载远程的文件系统到你本地的系统当中。

插件管理

Pathogen是第一个比较流行的插件管理工具。实际上它只是修改了 *runtimepath* （`:h 'rtp'`） 来引入所有放到该目录下的文件。你需要自己克隆插件的代码仓库到那个目录。

真正的插件管理工具会在 Vim 中提供帮助你安装或更新插件的命令。以下是一些常用的插件管理工具：

- dein
- plug
- vim-addon-manager
- vundle

多行编辑

这是一种可以同时输入多行连续文本的技术。参考这个示例。

用 `<c-v>` 切换到可视块模式。然后向下选中几行，按 `I` 或 `A` （译者注：大写字母，即 shift+i 或 shift+a）然后开始输入你想要输入的文本。

在刚开始的时候可能会有些迷惑，因为文本只出现在了当前编辑的行，只有在当前的插入动作结束后，之前选中的其它行才会出现插入的文本。

举一个简单的例子：`<c-v>3jItext<esc>`。

如果你要编辑的行长度不同，但是你想在他们后面追加相同的内容的话，可以试一下这个：`<c-v>3j$Atext<esc>`。

有时你可能需要把光标放到当前行末尾之后，默认情况下你是不可能做到的，但是可能通过设置 `virtualedit` 选项达到目的：

```
set virtualedit=all
```

设置之后 `$10l` 或 `90|` 都会生效，即使超过了行尾的长度。

详见 `:h blockwise-examples`。在开始的时候可能会觉得有些复杂，但是它很快就会成为你的第二天性的。

如果你想探索更有趣的事情，可以看看多光标

使用外部程序和过滤器

免责声明：Vim 是单线程的，因此在 Vim 中以前端进程执行其它的程序时会阻止其它的一切。当然你可以使用 Vim 程序接口，如 Lua，并且使用它的多线程支持，但是在那期间， Vim 的处理还是被阻止了。Neovim 添加了任务 API 解决了此问题。

（据说 Bram 正在考虑在 Vim 中也添加任务控制。如果你使用了较新版本的的 Vim ，可以看一下 `:helpgrep startjob`。）

使用 `:!` 启动一个新任务。如果你想列出当前工作目录下的所有文件，可以使用 `:!ls`。 用 `|` 来将结果通过管道重定向，如：`:!ls -l | sort | tail -n5`。

没有使用范围时（译者注：范围就是 `:` 和 `!` 之间的内容，`.` 表示当前行，`+4` 表示向下偏移 4 行，`$` 表示最末行等，多行时用 `,` 将它们分开，如 `.,$` 表示从当前行到末行），`:!` 会显示在一个可滚动的窗口中（译者注：在 GVim 和在终端里运行的结果稍有不同）。相反的，如果指定了范围，这些行会被过滤。这意味着它们会通过管道被重定向到过滤程序的

stdin，在处理后再通过过滤程序的 stdout 输出，用输出结果替换范围内的文本。例如：为接下来的 5 行文本添加行号，可以使用：

```
:.,+4!nl -ba -w1 -s' '
```

由于手动添加范围很麻烦， Vim 提供了一些辅助方法以方便的添加范围。如果需要经常带着范围的话，你可以在可见模式中先选择，然后再按 `:`（译者注：选中后再按 `!` 更方便）。还可以使用 `!` 来取用一个 motion 的范围，如 `!ipsort` （译者注：原文为 `!ip!sort` ，但经过实验发现该命令执行报错，可能是因为 Vim 版本的原因造成的，新版本使用 `ip` 选择当前段落后自动在命令后添加了 `!` ，按照作者的写法来看，可能之前的版本没有自动添加 `!` ）可以将当前段落的所有行按字母表顺序进行排序。

一个使用过滤器比较好的案例是Go 语言。它的缩进语法非常个性，甚至还专门提供了一个名为 `gofmt` 的过滤器来对 Go 语言的源文件进行正确的缩进。Go 语言的插件通常会提供一个名为 `:Fmt` 的函数，这个函数就是执行了 `:%!gofmt` 来对整个文件进行缩进。

人们常用 `:r !prog` 将 prog 程序的插入放到当前行的下面，这对于脚本来说是很不错的选择，但是在使用的过程中我发现 `!!ls` 更加方便，它会用输出结果替换当前行的内容。（译者注：前面命令中的 `prog` 只是个占位符，在实际使用中需要替换成其它的程序，如 `:r !ls`，这就与后面的 `!!ls` 相对应了，两者唯一的不同是第一个命令不会覆盖当前行内容，但是第二个命令会）

帮助文档：

```
:h filter
:h :read!
```

Cscope

Cscope 的功能比 ctags 要完善，但是只支持 C（通过设置 cscope.files 后同样支持 C++以及 Java）。

鉴于 Tag 文件只是知道某个符号是在哪里定义的，cscope 的数据库里的数据信息就多的多了：

- 符号是在哪里定义的?
- 符号是在哪里被使用的?

- 这个全局符号定义了什么?
- 这个变量是在哪里被赋值的?
- 这个函数在源文件的哪个位置?
- 哪些函数调用了这个函数?
- 这个函数调用了哪些函数?
- "out of space"消息是从哪来的?
- 在目录结构中当前的源文件在哪个位置?
- 哪些文件引用了这个头文件?

1. 构建数据库

在你项目的根目录执行下面的命令:

```
$ cscope -bqR
```

这条命令会在当前目录下创建三个文件: `cscope{,.in,.po}.out` 。把它们想象成你的数据库。

不幸的时 `cscope` 默认只分析 `*.[c|h|y|l]` 文件。如果你想在 Java 项目中使用 cscope， 需要这样做:

```
$ find . -name "*.java" > cscope.files
$ cscope -bq
```

2. 添加数据库

打开你新创建的数据库连接:

```
:cs add cscope.out
```

检查连接已经创建成功:

```
:cs show
```

(当然你可以添加多个连接。)

3. 查询数据库

```
:cs find <kind> <query>
```

如：`:cs find d foo` 会列出 `foo(...)` 调用的所有函数。

Kind	说明
s	**s**ymbol：查找使用该符号的引用
g	**g**lobal：查找该全局符号的定义
c	**c**alls：查找调用当前方法的位置
t	**t**ext：查找出现该文本的位置
e	**e**grep：使用 egrep 搜索当前单词
f	**f**ile：打开文件名
i	**i**ncludes：查询引入了当前文件的文件
d	**d**epends：查找当前方法调用的方法

推荐一些比较方便的映射，如：

```
nnoremap <buffer> <leader>cs :cscope find s  <c-r>=expand('<cword>')<cr><cr>
nnoremap <buffer> <leader>cg :cscope find g  <c-r>=expand('<cword>')<cr><cr>
nnoremap <buffer> <leader>cc :cscope find c  <c-r>=expand('<cword>')<cr><cr>
nnoremap <buffer> <leader>ct :cscope find t  <c-r>=expand('<cword>')<cr><cr>
nnoremap <buffer> <leader>ce :cscope find e  <c-r>=expand('<cword>')<cr><cr>
nnoremap <buffer> <leader>cf :cscope find f  <c-r>=expand('<cfile>')<cr><cr>
nnoremap <buffer> <leader>ci :cscope find i ^<c-r>=expand('<cfile>')<cr>$<cr>
nnoremap <buffer> <leader>cd :cscope find d  <c-r>=expand('<cword>')<cr><cr>
```

所以 `:tag` （或 `<c-]>`）跳转到标签定义的文件，而 `:cstag` 可以达到同样的目的，同时还会打开 cscope 的数据库连接。`'cscopetag'` 选项使得 `:tag` 命令自动的像 `:cstag` 一样工作。这在你已经使用了基于标签的映射时会非常方便。

帮助文档：`:h cscope`

MatchIt

由于 Vim 是用 C 语言编写的，因此许多功能都假设使用类似 C 语言的语法。默认情况下，如果你的光标在 `{` 或 `#endif`，就可以使用 `%` 跳转到与之匹配的 `}` 或 `#ifdef`。

Vim 自带了一个名为 matchit.vim 的插件，但是默认没有启用。启用后可以用 `%` 在 HTML 相匹配的标签或 VimL 的 if/else/endif 块之间进行跳转，它还带来了一些新的命令。

在 Vim 8 中安装

```
" vimrc
packadd! matchit
```

在 Vim 7 或者更早的版本中安装

```
"vimrc
runtime macros/matchit.vim
```

由于 matchit 的文档很全面，我建议安装以后执行一次下面的命令：

```
:!mkdir -p ~/.vim/doc
:!cp $VIMRUNTIME/macros/matchit.vim ~/.vim/doc
:helptags ~/.vim/doc
```

简短的介绍

至此这个插件已经可以使用了。 参考 `:h matchit-intro` 来获得支持的命令以及 `:h matchit-languages` 来获得支持的语言。

你可以很方便的定义自己的匹配对，如：

```
autocmd FileType python let b:match_words = '\<if\>:\
<elif\>:\<else\>'
```

之后你就可以在任何的 Python 文件中使用 `%` （向前）或 `g%` （向后）在这三个片断之间跳转了。

帮助文档:

```
:h matchit-install
:h matchit
:h b:match_words
```

技巧

跳至选择的区域另一端

在使用 `v` 或者 `V` 选择某段文字后，可以用 `o` 或者 `O` 按键跳至选择区域的开头或者结尾。

```
:h v_o
:h v_O
```

聪明地使用 n 和 N

`n` 与 `N` 的实际跳转方向取决于使用 `/` 还是 `?` 来执行搜索，其中 `/` 是向后搜索，`?` 是向前搜索。一开始我（原作者）觉得这里很难理解。

如果你希望 `n` 始终为向后搜索，`N` 始终为向前搜索，那么只需要这样设置:

```
nnoremap <expr> n  'Nn'[v:searchforward]
nnoremap <expr> N  'nN'[v:searchforward]
```

聪明地使用命令行历史

我（原作者）习惯用 `Ctrl` + `p` 和 `Ctrl` + `n` 来跳转到上一个/下一个条目。其实这个操作也可以用在命令行中，快速调出之前执行过的命令。

不仅如此，你会发现 `上` 和 `下` 其实更智能。如果命令行中已经存在了一些文字，我们可以通过按方向键来匹配已经存在的内容。比如，命令行中现在是 `:echo`，这时候我们按 `上`，就会帮我们补全成 `:echo "Vim rocks!"`（前提是，之前输入过这段命令）。

当然，Vim 用户都不愿意去按方向键，事实上我们也不需要去按，只需要设置这样的映射：

```
cnoremap <c-n> <down>
cnoremap <c-p> <up>
```

这个功能，我（原作者）每天都要用很多次。

智能 Ctrl-l

`Ctrl` + `l` 的默认功能是清空并「重新绘制」当前的屏幕，就和 `:redraw!` 的功能一样。下面的这个映射就是执行重新绘制，并且取消通过 `/` 和 `?` 匹配字符的高亮，而且还可以修复代码高亮问题（有时候，由于多个代码高亮的脚本重叠，或者规则过于复杂，Vim 的代码高亮显示会出现问题）。不仅如此，还可以刷新「比较模式」（请参阅 `:help diff-mode`）的代码高亮：

```
nnoremap <leader>l :nohlsearch<cr>:diffupdate<cr>:syntax sync fromstart<cr><c-l>
```

禁用错误报警声音和图标

```
set noerrorbells
set novisualbell
set t_vb=
```

请参阅 Vim Wiki: Disable beeping。

快速移动当前行

有时，我（原作者）想要快速把当前行上移或下移一行，只需要这样设置映射：

```
nnoremap [e  :<c-u>execute 'move -1-'. v:count1<cr>
nnoremap ]e  :<c-u>execute 'move +'. v:count1<cr>
```

这个映射，同样可以搭配数字使用，比如连续按下 `2` `]` `e` 就可以把当前行向下移动两行。

快速添加空行

```
nnoremap [<space>  :<c-u>put! =repeat(nr2char(10), v:count1)<cr>'[
nnoremap ]<space>  :<c-u>put =repeat(nr2char(10), v:count1)<cr>
```

设置之后，连续按下 `5` `[` `空格` 在当前行上方插入 5 个空行。

运行时检测

需要的特性：+profile

Vim 提供了一个内置的运行时检查功能，能够找出运行慢的代码。

`:profile` 命令后面跟着子命令来确定要查看什么。

如果你想查看所有的：

```
:profile start /tmp/profile.log
:profile file *
:profile func *
<do something in Vim>
<quit Vim>
```

Vim 不断地在内存中检查信息，只在退出的时候输出出来。（Neovim 已经解决了这个问题用 `:profile dump` 命令）

看一下 `/tmp/profile.log` 文件，检查时运行的所有代码都会被显示出来，包括每一行代码运行的频率和时间。

大多数代码都是用户不熟悉的插件代码，如果你是在解决一个确切的问题，
直接跳到这个日志文件的末尾，那里有 `FUNCTIONS SORTED ON TOTAL TIME` 和 `FUNCTIONS SORTED ON SELF TIME` 两个部分，如果某个

function 运行时间过长一眼就可以看到。

查看启动时间

感觉 Vim 启动的慢？到了研究几个数字的时候了：

```
vim --startuptime /tmp/startup.log +q && vim /tmp/startup.log
```

第一栏是最重要的因为它显示了**绝对运行时间**，如果在前后两行之间时间差有很大的跳跃，那么是第二个文件太大或者含有需要检查的错误的 VimL 代码。

NUL 符用新行表示

文件中的 NUL 符 （`\0`），在内存中被以新行（`\n`）保存，在缓存空间中显示为 `^@`。

更多信息请参看 `man 7 ascii` 和 `:h NL-used-for-Nul` 。

快速编辑自定义宏

这个功能真的很实用！下面的映射，就是在一个新的命令行窗口中读取某一个寄存器（默认为 `*`）。当你设置完成后，只需要按下 回车 即可让它生效。

在录制宏的时候，我经常用这个来更改拼写错误。

```
nnoremap <leader>m  :<c-u><c-r><c-r>='let @'. v:register .' = '. string(getreg(v:register))<cr><c-f><left>
```

只需要连续按下 leader m 或者 ″ leader m 就可以调用了。

请注意，这里之所以要写成 `<c-r><c-r>` 是为了确保 `<c-r>` 执行了。请参阅 `:h c_^R^R`

快速跳转到源(头)文件

这个技巧可以用在多种文件类型中。当你从源文件或者头文件中切换到其他文件的时候，这个技巧可以设置「文件标记」（请参阅 `:h marks`），然后你就可以通过连续按下 `'` `C` 或者 `'` `H` 快速跳转回去（请参阅 `:h 'A`）。

```
autocmd BufLeave *.{c,cpp} mark C
autocmd BufLeave *.h       mark H
```

注意：由于这个标记是设置在 viminfo 文件中，因此请先确认 `:set viminfo?` 中包含了 `:h viminfo-'`。

在 GUI 中快速改变字体大小

印象中，我（原作者）记得一下代码是来自 tpope's 的配置文件：

```
command! Bigger  :let &guifont = substitute(&guifont, '\d\+$', '\=submatch(0)+1', '')
command! Smaller :let &guifont = substitute(&guifont, '\d\+$', '\=submatch(0)-1', '')
```

根据模式改变光标类型

我（原作者）习惯在普通模式下用块状光标，在插入模式下用条状光标（形状类似英文 "I" 的样子），然后在替换模式中使用下划线形状的光标。

```
if empty($TMUX)
  let &t_SI = "\<Esc>]50;CursorShape=1\x7"
  let &t_EI = "\<Esc>]50;CursorShape=0\x7"
  let &t_SR = "\<Esc>]50;CursorShape=2\x7"
else
  let &t_SI = "\<Esc>Ptmux;\<Esc>\<Esc>]50;CursorShape=1\x7\<Esc>\\"
  let &t_EI = "\<Esc>Ptmux;\<Esc>\<Esc>]50;CursorShape=0\x7\<Esc>\\"
  let &t_SR = "\<Esc>Ptmux;\<Esc>\<Esc>]50;CursorShape=2\x7\<Esc>\\"
endif
```

原理很简单，就是让 Vim 在进入和离开插入模式的时候，输出一些序列，请参考 escape sequence。Vim 与终端之间的中间层，比如 tmux 会处理并执行上面的代码。

但上面这个还是有一个缺点的。终端环境的内部原理不尽相同，对于序列的处理方式也稍有不同。因此，上面的代码可能无法在你的环境中运行。甚至，你的运行环境也有可能不支持其他光标形状，请参阅你的 Vim 运行环境的文档。

好消息是，上面这个代码，可以在 iTerm2 中完美运行。

防止水平滑动的时候失去选择

如果你选中了一行或多行，那么你可以用 < 或 > 来调整他们的缩进。但在调整之后就不会保持选中状态了。

你可以连续按下 g v 来重新选中他们，请参考 `:h gv`。因此，你可以这样来配置映射：

```
xnoremap <  <gv
xnoremap >  >gv
```

设置好之后，在可视模式中使用 `>>>>>` 就不会再出现上面提到的问题了。

选择当前行至结尾，排除换行符

在 Vim 里，我们可以同过 `v$` 选择当前行至结尾，但此时会把最后一个换行符也选中，通常需要按额外的 `h` 来取消最后选中最后一个换行符号。Vim 提供了一个 `g_` 快捷键，可以移动光标至最后一个非空字符。因此，为达到次效果，可以使用 `vg_`。当然，如果觉得按三个键比较麻烦，可以添加一个映射：

```
nnoremap L g_
```

这样就可以通过 `vL` 达到一样的效果了。

重新载入保存文件

通过自动命令，你可以在保存文件的同时触发一些其他功能。比如，如果这个文件是一个配置文件，那么就重新载入；或者你还可以对这个文件进行代码风格检查。

```
autocmd BufWritePost $MYVIMRC source $MYVIMRC
autocmd BufWritePost ~/.Xdefaults call system('xrdb ~/.Xdefaults')
```

更加智能的当前行高亮

我（原作者）很喜欢「当前行高亮」（请参阅 `:h cursorline`）这个功能，但我只想让这个效果出现在当前窗口，而且在插入模式中关闭这个效果：

```
autocmd InsertLeave,WinEnter * set cursorline
autocmd InsertEnter,WinLeave * set nocursorline
```

更快的关键字补全

关键字补全（`<c-n>` 或 `<c-p>`）功能的工作方式是，无论 `'complete'` 设置中有什么，它都会尝试着去补全。这样，一些我们用不到的标签也会出现在补全列表中。而且，它会扫描很多文件，有时候运行起来非常慢。如果你不需要这些，那么完全可以像这样把它们禁用掉：

```
set complete-=i   " disable scanning included files
set complete-=t   " disable searching tags
```

改变颜色主题的默认外观

如果你想让状态栏在颜色主题更改后依然保持灰色，那么只需要这样设置：

```
autocmd ColorScheme * highlight StatusLine ctermbg=darkgray cterm=NONE guibg=darkgray gui=NONE
```

同理，如果你想让某一个颜色主题（比如 "lucius"）的状态栏为灰色（请使用 `:echo color_name` 来查看当前可用的所有颜色主题）：

```
autocmd ColorScheme lucius highlight StatusLine
ctermbg=darkgray cterm=NONE guibg=darkgray gui=NONE
```

命令

下面的命令都比较有用，最好了解一下。用 `:h :<command name>` 来了解更多关于它们的信息，如：`:h :global`。

:global 和 :vglobal - 在所有匹配行执行命令

在所有符合条件的行上执行某个命令。如：`:global /regexp/ print` 会在所有包含 "regexp" 的行上执行 `print` 命令（译者注：regexp 有正则表达式的意思，该命令同样支持正则表达式，在所有符合正则表达式的行上执行指定的命令）。

趣闻：你们可能都知道老牌的 grep 命令，一个由 Ken Thompson 编写的过滤程序。它是干什么用的呢？它会输出所有匹配指定正则表达式的行！现在猜一下 `:global /regexp/ print` 的简写形式是什么？没错！就是 `:g/re/p`。 Ken Thompsom 在编写 grep 程序的时候是受了 vi `:global` 的启发。（译者注：[https://robots.thoughtbot.com/how-grep-got-its-name)](https://robots.thoughtbot.com/how-grep-got-its-name)

既然它的名字是 `:global`，理应仅作用在所有行上，但是它也是可以带范围限制的。假设你想使用 `:delete` 命令删除从当前行到下一个空行（由正则表达式 `^$` 匹配）范围内所有包含 "foo" 的行：

```
:,/^$/g/foo/d
```

如果要在所有 *不* 匹配的行上执行命令的话，可以使用 `:global!` 或是它的别名 `:vglobal`（V 代表的是 inVerse）。

:normal 和 :execute - 脚本梦之队

这两个命令经常在 Vim 的脚本里使用。

借助于 `:normal` 可以在命令行里进行普通模式的映射。如：`:normal! 4j` 会令光标下移 4 行（由于加了"!"，所以不会使用自定义的映射 "j"）。

需要注意的是 `:normal` 同样可以使用范围数（译者注：参考 `:h range` 和 `:h :normal-range` 了解更多），故 `:%norm! Iabc` 会在所有行前加上 "abc"。

借助于 `:execute` 可以将命令和表达式混合在一起使用。假设你正在编辑一个 C 语言的文件，想切换到它的头文件：

```
:execute 'edit' fnamemodify(expand('%'), ':r') . '.h'
```

（译者注：头文件为与与源文件同名但是扩展名为 `.h` 的文件。上面的命令中 expand 获得当前文件的名称，fnamemodify 获取不带扩展名的文件名，再连上 '.h' 就是头文件的文件名了，最后在使用 edit 命令打开这个头文件。）

这两个命令经常一起使用。假设你想让光标下移 n 行：

```
:let n = 4
:execute 'normal!' n . 'j'
```

重定向消息

许多命令都会输出消息，`:redir` 用来重定向这些消息。它可以将消息输出到文件、寄存器或是某个变量中。

```
" 将消息重定向到变量 `neatvar` 中
:redir => neatvar
" 打印所有寄存器的内容
:reg
" 结束重定向
:redir END
" 输出变量
:echo neatvar
" 恶搞一下，我们把它输出到当前缓冲区
:put =neatvar
```

再 Vim 8 中，可以更简单的方式即位：

```
:put =execute('reg')
```

（译者注：原文最后一条命令是 `:put =nicevar` 但是实际会报变量未定义的错误）
（实测 neovim/vim8 下没问题）

帮助文档：`:h :redir`

调试

常规建议

如果你遇到了奇怪的行为，尝试用这个命令重现它：

```
vim -u NONE -N
```

这样会在不引用 vimrc（默认设置）的情况下重启 vim，并且在 **nocompatible** 模式下（使用 vim 默认设置而不是 vi 的）。（搜索 `:h --noplugin` 命令了解更多启动加载方式）

如果仍旧能够出现该错误，那么这极有可能是 vim 本身的 bug，请给 vim_dev 发送邮件反馈错误，多数情况下问题不会立刻解决，你还需要进一步研究

许多插件经常会提供新的（默认的/自动的）操作。如果在保存的时候发生了，那么请用 `:verb au BufWritePost` 命令检查潜在的问题

如果你在使用一个插件管理工具，将插件行注释调，再进行调试。

问题还没有解决？如果不是插件的问题，那么肯定是你的自定义的设置的问题，可能是你的 options 或 autocmd 等等。

到了一行行代码检查的时候了，不断地排除缩小检查范围知道你找出错误，根据二分法的原理你不会花费太多时间的。

在实践过程中，可能就是这样，把 `:finish` 放在你的 **vimrc** 文件中间，Vim 会跳过它之后的设置。如果问题还在，那么问题就出在`:finish`之前的设置中，再把`:finish`放到前一部分设置的中间位置。否则问题就出现在它后面的半部分设置，那么就把`:finish`放到后半部分的中间位置。不断的重复即可找到。

调整日志等级

Vim 现在正在使用的另一个比较有用的方法是增加 debug 信息输出详细等级。现在 Vim 支持 9 个等级，可以用`:h 'verbose'`命令查看。

```
:e /tmp/foo
:set verbose=2
:w
:set verbose=0
```

这可以显示出所有引用的文件、没有变化的文件或者各种各样的作用于保存的插件。

如果你只是想用简单的命令来提高等级，也是用 `:verbose` ，放在其他命令之前，通过计数来指明等级，默认是 1.

```
:verb set verbose
"  verbose=1
:10verb set verbose
"  verbose=10
```

通常用等级 1 来显示上次从哪里设置的选项

```
:verb set ai?
"      Last set from ~/.vim/vimrc
```

一般等级越高输出信息月详细。但是不要害怕，亦可以把输出导入到文件中：

```
:set verbosefile=/tmp/foo | 15verbose echo "foo" |
vsplit /tmp/foo
```

你可以一开始的时候就打开 verbosity，用 `-V` 选项，它默认设置调试等级为 10。 例如：`vim -V5`

查看启动日志

查看运行时日志

Vim 脚本调试

如果你以前使用过命令行调试器的话，对于`:debug`命令你很快就会感到熟悉。

只需要在任何其他命令之前加上`:debug`就会让你进入调试模式。也就是，被调试的 Vim 脚本会在第一行停止运行，同时该行会被显示出来。

想了解可用的 6 个调试命令，可以查阅`:h >cont`和阅读下面内容。需要指出的是，类似 gdb 和其他相似调试器，调试命令可以使用它们的简短形式：`c`、 `q`、`n`、`s`、 `i`和 `f`。

除了上面的之外，你还可以自由地使用任何 Vim 的命令。比如，`:echo myvar`，该命令会在当前的脚本代码位置和上下文上被执行。

只需要简单使用`:debug 1`，你就获得了REPL调试特性。

当然，调试模式下是可以定义断点的，不然的话每一行都去单步调试就会十分痛苦。（断点之所以被叫做断点，是因为运行到它们的时候，运行就会停止下来。因此，你可以利用断点跳过自己不感兴趣的代码区域）。请查阅`:h :breakadd`、 `:h :breakdel`和 `:h :breaklist`获取更多细节。

假设你需要知道你每次在保存一个文件的时候有哪些代码在运行：

```
:au BufWritePost
" signify  BufWritePost
"     *         call sy#start()
:breakadd func *start
:w
" Breakpoint in "sy#start" line 1
" Entering Debug mode.  Type "cont" to continue.
" function sy#start
" line 1: if g:signify_locked
>s
" function sy#start
" line 3: endif
>
" function sy#start
" line 5: let sy_path = resolve(expand('%:p'))
>q
:breakdel *
```

正如你所见，使用`<cr>`命令会重复之前的调试命令，也就是在该例子中的`s`命令。

`:debug`命令可以和verbose选项一起使用。

语法文件调试

语法文件由于包含错误的或者复制的正则表达式，常常会使得 Vim 的运行较慢。如果 Vim 在编译的时候包含了`+profile` feature特性，就可以给用户提供一个超级好用的`:syntime`命令。

```
:syntime on
" 多次敲击<c-l>来重绘窗口，这样的话就会使得相应的语法规则被重新应用一次
:syntime off
:syntime report
```

输出结果包含了很多的度量维度。比如，你可以通过结果知道哪些正则表达式耗时太久需要被优化；哪些正则表达式一直在别使用但重来没有一次成功匹配。

请查阅`:h :syntime`。

杂项

附加资源

资源名称	简介
七个高效的文本编辑习惯	作者：Bram Moolenaar（即 Vim 的作者）
七个高效的文本编辑习惯 2.0（PDF 版）	同上
IBM DeveloperWorks: 使用脚本编写 Vim 编辑器	Vim 脚本编写五辑
《漫漫 Vim 路》	使用魔抓定制 Vim 插件
《 Vim 实践 (第 2 版)》	轻取 Vim 最佳书籍
Vimcasts.org	Vim 录屏演示
为什么是个脚本都用 vi?	常见误区释疑
你不爱 vi，所以你不懂 Vim	简明,扼要,准确的干货

Vim 配置集合

目前，网上有很多流行 Vim 配置集合，对于 Vim 配置集合，个人认为有利有弊。
对于维护的比较好的配置，比如 SpaceVim 还是值得尝试的，可以节省很多自行配置的时间。
当然，网上还有很多其他很流行的配置，比如：

- k-vim
- amix's vimrc
- janus

常见问题

编辑小文件时很慢

有两个因素对性能影响非常大：

1. 过于复杂的 **正则表达式** 。尤其是 Ruby 的语法文件，以前会造成性能下降。（见调试语法文件）
2. **屏幕重绘** 。有一些功能会强制重绘所有行。

典型肇事者	原因	解决方案
`:set cursorline`	会导致所有行重绘	`:set nocursorline`
`:set cursorcolumn`	会导致所有行重绘	`:set nocursorcolumn`
`:set relativenumber`	会导致所有行重绘	`:set norelativenumber`
`:set foldmethod=syntax`	如果语法文件已经很慢了，这只会变得更慢	`:set foldmethod=manual`，`:set foldmethod=marker` 或者使用快速折叠插件
`:set synmaxcol=3000`	由于内部表示法，Vim 处理比较长的行时会有问题。让它高亮到 3000 列.....	`:set synmaxcol=200`
matchparen.vim	Vim 默认加载的插件，用正则表达式查找配对的括号	禁用插件：`:h matchparen`

注意：只有在你真正遇到性能问题的时候才需要做上面的调整。在大多数情况下使用上面提到的选项是完全没有问题的。

编辑大文件的时候很慢

Vim 处理大文件最大的问题就是它会一次性读取整个文件。这么做是由于缓冲区的内部机理导致的（在 vim_dev 中讨论）。

如果只是想查看的话，`tail hugefile | vim -` 是一个不错的选择。

如果你能接受没有语法高亮，并且禁用所有插件和设置的话，使用：

```
$ vim -u NONE -N
```

这将会使得跳转变快很多，尤其是省去了基于很耗费资源的正则表达式的语法高亮。你还可以告诉 Vim 不要使用交换文件和 viminfo 文件，以避免由于写这些文件而造成的延时：

```
$ vim -n -u NONE -i NONE -N
```

简而言之，尽量避免使用 Vim 写过大的文件。

持续粘贴（为什么我每次都要设置 'paste' 模式）

持续粘贴模式让终端模拟器可以区分输入内容与粘贴内容。

你有没有遇到过往 Vim 里粘贴代码之后被搞的一团糟?

这在你使用 `cmd+v`、`shirt-insert`、`middle-click` 等进行粘贴的时候才会发生。
因为那样的话你只是向终端模拟器扔了一大堆的文本。
Vim 并不知道你刚刚是粘贴的文本，它以为你在飞速的输入。
于是它想缩进这些行但是失败了。

这明显不是个问题，如果你用 Vim 的寄存器粘贴，如：`"+p`，这时 Vim 就知道了你在粘贴，就不会导致格式错乱了。

使用 `:set paste` 就可以解决这个问题正常进行粘贴。见 `:h 'paste'` 和 `:h 'pastetoggle'` 获取更多信息。

如果你受够了每次都要设置 `'paste'` 的话，看看这个能帮你自动设置的插件：bracketed-paste。

点此查看该作者对于这个插件的更多描述。

Neovim 尝试把这些变得更顺畅，如果终端支持的话，它会自动开启持续粘贴模式，无须再手动进行切换。

在终端中按 ESC 后有延时

如果你经常使用命令行，那么肯定要接触 *终端模拟器*，如 xterm、gnome-terminal、iTerm2 等等（与实际的终端不同）。

终端模拟器与他们的祖辈一样，使用 转义序列 （也叫 *控制序列*）来控制光标移动、改变文本颜色等。转义序列就是以转义字符开头的 ASCII 字符串（用脱字符表示法表示成 `^[` ）。当遇到这样的字符串后，终端模拟器会从终端信息数据库中查找对应的动作。

为了使用问题更加清晰，我会先来解释一下什么是映射超时。在映射存在歧义的时候就会产生映射超时：

```
:nnoremap ,a :echo 'foo'<cr>
:nnoremap ,ab :echo 'bar'<cr>
```

上面的例子中两个映射都能正常工作，但是当输入 `,a` 之后，Vim 会延时 1 秒，因为它要确认用户是否还要输入那个 `b`。

转义序列会产生同样的问题：

- `<esc>` 作为返回普通模式或取消某个动作的按键而被大量使用
- 光标键使用转义序列进行的编码
- Vim 期望 `Alt` （也叫作 *Mate Key* ）会发送一个正确的 8-bit 编码的高位，但是许多终端模拟器并不支持这个（也可能默认没有启用），而只是发送一个转义序列作为代替。

你可以这样测试上面所提到的事情： `vim -u NONE -N` 然后输入 `i<c-v><left>` ，你会看到一个以 `^[` 开头的字符串，表明这是一个转义序列，`^[` 就是转义字符。

简而言之，Vim 在区分录入的 `<esc>` 和转义序列的时候需要一定的时间。

默认情况下，Vim 用 `:set timeout timeoutlen=1000`，就是说它会用 1 秒的时间来区分有歧义的映射 *以及* 按键编码。这对于映射来说是一个比较合理的值，但是你可以自行定义按键延时的长短，这是解决该问题最根本的办法：

```
set timeout             " for mappings
set timeoutlen=1000     " default value
set ttimeout            " for key codes
set ttimeoutlen=10      " unnoticeable small value
```

在 `:h ttimeout` 里你可以找到一个关于这些选项之间关系的小表格。

而如果你在 tmux 中使用 Vim 的话，别忘了把下面的配置加入到你的 `~/.tmux.conf` 文件中：

```
set -sg escape-time 0
```

无法重复函数中执行的搜索

- 在命令中的搜索（`/`、`:substitute` 等）内容会改变“上次使用的搜索内容”。（它保存在 `/` 寄存器中，用 `:echo @/` 可以输出它里面的内容）
- 简单的文本变化可以通过 `.` 重做。（它保存在 `.` 寄存器，用 `:echo @.` 可以输出它的内容）

而在你在函数中进行这些操作的时候，一切就会变得不同。因此你不能用 N/n 查找某个函数刚刚查找的内容，也不能重做函数中对文本的修改。

帮助文档：`:h function-search-undo`。

进阶阅读

- Vim 插件开发指南
- 常用插件列表

加入我们

可以协助我们核对翻译，或者从章节列表中认领章节进行翻译。

参考资料

- Nifty Little Nvim Techniques to Make My Life Easier -- Series 1