

深入 Vim 7.2

Kim Schulz

https://github.com/wuzhouhui/hacking_vim

2016 年 11 月 20 日

目录

前言	v
第一章 开始	1
1.1 获取 Vim	1
1.2 vi, Vim, 及其朋友	2
1.2.1 vi	2
1.2.2 STEVIE	2
1.2.3 Elvis	2
1.2.4 nvi	3
1.2.5 Vim	4
1.2.6 Vile	4
1.2.7 兼容性	5
1.3 博爱的 Vim	6
1.4 公共术语	6
1.5 小结	6
第二章 定制 Vim	7
2.1 配置文件的存放位置	7
2.2 更改字体	9
2.3 修改配色方案	10
2.3.1 个性化高亮	10
2.4 更丰富的状态行	13
2.5 切换菜单与工具条	15
2.6 添加自定义菜单与工具条按钮	15
2.7 修改标签页	18
2.8 工作区定制	22
2.8.1 为光标添加视觉效果	22
2.8.2 添加行号	23
2.8.3 拼写检查	24
2.8.4 添加帮助性的工具提示	25
2.8.5 使用缩写	28
2.8.6 修改按键绑定	30
2.9 小结	32

第三章 快速导航	33
3.1 在文件内更快地导航	33
3.1.1 基于上下文的导航	33
3.1.2 在长行内导航	37
3.2 在 Vim 帮助中快速地导航	37
3.3 在多个缓冲区中更快地导航	38
3.4 快速打开引用过的文件	39
3.5 搜索即可得到	40
3.5.1 在当前文件内搜索	40
3.5.2 在多个文件内搜索	41
3.5.3 搜索帮助系统	42
3.6 标记位置	43
3.6.1 可见的标记 — 使用符号	43
3.6.2 隐藏的标记	45
3.7 小结	46
第四章 助推器	47
4.1 使用模版	47
4.1.1 使用模版文件	48
4.1.2 把缩写作为模版	49
4.1.3 snipMate 脚本	50
4.2 Tag List	51
4.2.1 更便捷的 taglist 导航	53
4.2.2 taglist 的其他用法	54
4.3 使用自动补全	54
4.3.1 已知单词的自动补全	54
4.3.2 使用字典的自动补全	55
4.3.3 omnicompletion	56
4.3.4 多合一补全	58
4.4 宏录制	59
4.5 使用会话	61
4.5.1 简单的会话使用	61
4.5.2 满足个人的会话需求	63
4.5.3 会话与项目管理	64
4.6 寄存器与撤消分支	65
4.6.1 使用寄存器	66
4.6.2 撤消分支	68
4.7 折叠	71
4.7.1 提取大纲	74
4.7.2 使用 vimdiff 比较差异	74
4.7.3 在 vimdiff 中导航	76

4.7.4	使用 diff 跟踪变化	77
4.8	打开任意位置的文件	77
4.8.1	更快的远程文件编辑	79
4.9	小结	79
第五章	格式化进阶	80
5.1	格式化文本	80
5.1.1	文本分段	80
5.1.2	对齐文本	82
5.1.3	标记标题	83
5.1.4	创建列表	84
5.2	格式化代码	86
5.2.1	Autoindent	86
5.2.2	Smartindent	86
5.2.3	cindent	86
5.2.4	Indentexpr	87
5.2.5	代码块快速格式化	87
5.2.6	自动格式化粘贴的代码	89
5.3	使用外部格式化工具	90
5.3.1	Indent	90
5.3.2	Berkeley Par	90
5.3.3	Tidy	91
5.4	小结	92
第六章	Vim 脚本基础	93
6.1	语法高亮方案	93
6.1.1	第一个语法高亮文件	93
6.2	区域高亮	94
6.2.1	配色方案与语法高亮	97
6.3	使用脚本	97
6.3.1	脚本类型	98
6.3.2	安装脚本	98
6.3.3	卸载脚本	99
6.4	脚本开发	99
6.4.1	脚本开发基础	100
6.5	小结	114
第七章	Vim 脚本进阶	115
7.1	脚本结构	115
7.1.1	脚本头部	115
7.1.2	脚本加载检查	116
7.1.3	脚本配置	116

7.1.4	按键映射	117
7.1.5	函数	118
7.1.6	一个完整的脚本	119
7.2	脚本开发技术	120
7.2.1	Gvim 或 Vim	120
7.2.2	操作系统类型	121
7.2.3	Vim 的版本	121
7.2.4	打印很长的行	122
7.3	调试 Vim 脚本	123
7.4	发布 Vim 脚本	125
7.4.1	制作 Vimball	125
7.5	注意文档	126
7.6	使用外部解释器	128
7.6.1	Perl	129
7.6.2	Python	130
7.6.3	Ruby	131
7.7	小结	132
附录 A	无所不能的 Vim	134
A.1	Vim 游戏	134
A.1.1	生命游戏	134
A.1.2	贪吃蛇	134
A.1.3	魔方	135
A.1.4	井字棋	135
A.1.5	扫雷	136
A.1.6	推箱子	136
A.1.7	俄罗斯方块	136
A.2	集成开发环境	137
A.3	邮件程序	139
A.4	聊天	139
A.5	Twitter 客户端	140
附录 B	Vim 配置管理	142
B.1	保持 vimrc 整洁的技巧	142
B.2	vimrc 配置系统	143
B.3	在线存放 vimrc	145
附录 C	索引	147

前言

1

在计算机发展的早期, 系统资源非常有限, 开发人员必须想尽办法优化他们的应用程序, 文本编辑器也是如此. Vim 是当时最流行的编辑器之一, 因为系统资源方面的限制, 它被优化得近乎完美.

从那以后, 计算机技术得到了快速的发展, 虽然系统资源已经没有以前那么紧张了, 但是 Vim 仍然坚持最初的原则.

乍看起来, Vim 好像并没有什么值得称道的地方, 但是, 如果你能透过它简单的用户界面看到其本质, 就会明白为什么到了如今这个年代, Vim 仍然是众多用户最喜爱的编辑器.

它几乎囊括了你想要的任何一个功能, 即使有所遗漏, 也可以通过插件或脚本来实现. 正是由于其出色的灵活性, 使得它成为了众多任务的理想工具, 以及世界上最先进的编辑器之一.

每天都有大量的新用户加入 Vim 社区, 并开始用 Vim 处理他们的日常工作. 虽然有时候使用起来比较复杂, 但是与其他编辑器相比, 人们还是更愿意选择 Vim, 本书就是为这些用户而写.

通过阅读本书, 用户可以更加得心应手地使用 Vim, 从而提高工作效率. 他们得到的不仅仅是一个优化的编辑器, 还有优化的工作流程. 本书帮助用户更加自如地使用 Vim, 从把它当作一个简单的文本编辑器开始, 一直到其他日常工作, Vim 都可以胜任.

祝你阅读愉快!

本书主要讨论什么

2

第一章: 开始, 介绍 Vim 和它的几个比较有名的亲戚, 并简要介绍它们和 vi 的关系及其历史.

第二章: 定制 Vim, 介绍如何配置 Vim, 使得它更符合用户的个人需求. 主要介绍如何修改字体, 配色方案, 状态行, 菜单, 与工具条.

第三章: 快速导航, 介绍一些在多个文件中快速导航的方法, 通过这些方法, 读者可以轻松地在多个文件之间穿梭.

第四章: 助推器, 介绍 Vim 的若干特性, 描述了模版, 自动补全, 拆叠, 会话, 和寄存器的使用方法.

第五章: 格式化进阶, 讨论如何对文本和代码进行格式化. 它还介绍了如何借助外部工具, 使得 Vim 更加完美.

第六章: Vim 脚本基础, 这一章是为那些想通过脚本来扩展 Vim 功能的人而准备的, 它介绍了脚本的基础知识, 经过这一章的学习, 读者应该能够写出一些简单的脚本.

第七章: Vim 脚本进阶, 在第六章的基础上, 再继续讲一些高级的脚本知识, 包括如何在 Vim 脚本中使用外部的脚本语言.

附录 A: 无所不能的 Vim, 提供了一张游戏列表, 它们都是用 Vim 脚本开发的, 这一章还简单讨论了如何使用脚本实现聊天和邮件工具, 另外还介绍了如何将 Vim 作为一个 IDE 使用.

附录 B: Vim 配置管理, 展示如何通过在线副本来更好地管理和获取 Vim 配置文件.

为了阅读本书,你还需要什么

最近十年, Vim 已经发展成一个功能非常丰富的编辑器,这同时意味着最新版的某些功能,旧版可能并不支持。

Vim 已经移植到了多种平台中,但并不是所有的功能对任何一种平台来说都是可用的。这主要是因为有些功能使用了和操作系统密切相关的特性,而这些特性在其他平台中可能并不提供。

本书只关注 Vim 使用最广泛的两种平台: Linux 与 Microsoft Windows。由于 Linux 符合 Unix 标准,所以同样的结论在其他类 Unix 系统中仍然成立。



读者可以在 www.vim.org 上找到最新版的 Vim 源代码与二进制包。如果读者使用的是 Linux,那么系统中很可能已经安装了 Vim,而且是默认的编辑器。

3

本书的目标读者

如果读者已经是一位 Vim 用户,而且想要学习更高级的技巧,那么这本书就是为你而写的。本书致力于帮助读者从一位 Vim 中级用户,上升为高级用户。

Vim 新手?

虽然本书假设读者已经具备了 Vim 的基本知识,但即使是新手,本书也有一定的参考价值。

如果读者担心自己所掌握的 Vim 知识不足以阅读此书,那么我建议你可以先看一下 Vimtutor,在安装 Vim 时,会顺带安装这个 Vim 入门教程。

运行 Vimtutor,然后跟着它学习 Vim 的基本使用方法,这大概需要 30 分钟。

Vimtutor 支持多国语言,在启动程序时,可以通过指定国家代码来选择一种语言,国家代码由 2 个字母组成,比如 ca, cs, de, el, eo, es, fr, hr, hu, it, ja, no, ro, 和 zh。例如,如果想要阅读德语版的 Vimtutor,只需要在命令行执行 `vimtutor de`。

排版约定

本书用不同风格的文本来表示不同种类的信息,这里有一些例子。

包含在正文中的代码会这样显示:“键入 `:help 'statusline'`,就可以打开 Vim 帮助系统,查看与状态行有关的全部帮助信息。”

代码块则是:

```
function! InfoGuiTooltip()  
    "get window count  
    let wincount = tabpagewinnr(tabpagenr(), '$')  
    let bufferlist=''  
    "get name of active buffers in windows  
    for i in tabpagebuflist()  
        let bufferlist .= '['.fnamemodify(bufname(i),':t').'] '  
    endfor  
endfunction
```

4

```
endfor
return bufname($). ' windows: '.wincount.' ' .bufferlist ' '
endfunction
```

命令模式的输入或输出写成:

```
:amenu icon=/path/to/icon/myicon.png ToolBar.Bufferlist :buffers<cr>
```

新的术语^① 或 **重点内容**加粗显示. 读者在屏幕上看到的内容, 比如菜单或对话框中的信息, 会显示成: “这个命令在所有行的开头查找单词 **Error** (用脱字符 ^ 标记)”.



这种文本框用来显示警告或重要的提示.



这种文本框用来显示技巧和诀窍.

读者反馈

对于读者的反馈我们总是非常欢迎. 请把你关于这本书的想法告诉我们 — 不管是优点还是缺点, 读者的反馈永远是我们不断进步的源泉.

反馈信息请发送到 feedback@packtpub.com, 并在邮件的主题中写上相关的书名.

如果读者想向我们推荐书籍, 请登陆 www.packtpub.com, 在表单 SUGGEST A TITLE 中填写相关的信息, 或者发送邮件到 suggest@packtpub.com.

如果你对某个主题很有研究, 并且对写作也很感兴趣, 请阅读 www.packtpub.com/authors 的相关内容.

5

客户支持

既然读者买了 Packt 出版的书籍, 那么我们会尽最大的努力让你觉得物有所值.



下载书中的示例源代码:

登陆 http://www.packtpub.com/files/code/0509_Code.zip, 下载书中的示例源代码, 下载的文件中含有使用方法.

勘误

我们已经尽了最大的努力来保证内容的正确性, 如果读者在书中发现了错误 — 这个错误可能出现在正文中, 也可能出现在代码中 — 请把错误反馈给我们. 通过反馈错误, 可以帮助其他读者更顺利地阅读, 也可以帮助我们提升后续版本的质量. 无论读者发现了什么类型的错误, 请登陆到 <http://www.packtpub.com/support>, 选择对应的书籍, 点击 let us know, 然后输入相关的勘误信息. 一旦勘误通过了校验, 你的提交信息就会被接受, 勘误也会上传到网页中, 或者加到书籍勘误列表, 这个列表可以在书籍的 Errata 部分找到. 登陆 <http://www.packtpub.com/support>, 选择对应的书籍, 读者就可以找到该书所有的已知勘误.

^①在中文版中并没有这样做 — 译者注

版权声明

因特网物品的版权一直是一个让人头疼的问题. Packt 对版权和授权的保护向来非常重视, 态度也很坚决. 如果读者怀疑有人在因特网上非法复制我们的作品 — 无论是什么形式 — 请把相关的网址或网站名称发给我们 (copyright@packtpub.com), 这样我们才能快速地追回损失. 希望读者能和我们一起尊重作者的辛勤付出, 这样我们才能继续为你奉献精彩的作品.

6

答疑

无论你对书籍有什么疑问, 都可以联系我们: questions@packtpub.com, 我们会尽最大的努力来回答你的问题.

第一章 开始

7

Vim (Vi IMproved) 编辑器最早由 Bram Moolenaar 于 1991 年 11 月发布, 当时只是作为 Unix vi 编辑器的 Amiga 平台克隆版.

一年后, Unix 平台的 Vim 发布, 接着, 它迅速成为了 vi 的替代版本.

由于宽松的授权和丰富的功能, 在开源社区的帮助下, Vim 逐渐流行起来. 越来越多的 Linux 发行版开始用 Vim 替换 vi. 虽然许多用户认为他们使用的是 vi (如果他们是通过执行命令 vi 来打开编辑的话), 可实际上打开的是 Vim (命令 vi 已经被 vim 的链接替换掉, 所以经常会有人误以为 vi 和 Vim 是同一个程序).

在九十年代后期, vi 在编辑器之战中所输掉的劣势, 重新又被 Vim 给赢了回来, 编辑器之战指的是 vi 和 Emacs 之间的斗争. Bram 为 Vim 扩充了许多新特性, 而这些特性原本被 Emacs 党利用, 作为论证 Vim/vi 不如 Emacs 的论据, 即使如此, Bram 仍然没有忘记先人开发 vi 的初衷.

如今, Vim 已经是一个功能丰富, 定制性强, 受人欢迎的编辑器. 它支持超过 200 种语言的语法高亮, 自动补全, 折叠, 撤消/重做, 多重缓冲区/窗口/标签, 以及其他特性.

本章主要介绍

- 如何获取与安装 Vim 编辑器
- Vim 编辑器家族
- Vim 的发布许可证
- 本书使用的公共术语

8

1.1 获取 Vim

读者也许对 Vim 有了一定的了解, 而且也使用了一段时间, 然而, 如果你还没有使用过 Vim, 那么最好趁现在这个时候, 在自己的系统中安装 Vim.

可以从网站 <http://www.vim.org> 下载到 Vim 的最新版.



本书主要讨论 Vim 7.2, 如果用户所用的版本比较老, 请不要担心, 可以随时更新到最新版.

如果读者的操作系统是 Microsoft Windows, 只需要双击运行下载的 *.exe 文件, 就可以开始安装过程. 安装完毕后, 在“开始”菜单中就会出现一个指向 gVim 的快捷键.

如果是 Linux, 那么安装方式取决于读者所使用的 Linux 发行版. 如今, 在大多数发行版中已经预装了 Vim, 如果没有, 具体的安装方法请参考发行版的软件包管理器 (例如, Debian 的软件包管理器是 Aptitude, Mandriva 是

urpmi, Ubuntu 的是 Synaptics)。如果系统中没有软件包管理器,还可以从上面所提供的网站中下载 Vim 的源代码,手工编译安装,具体的编译安装方法可以看一下源码包中的 readme 文件。

1.2 vi, Vim, 及其朋友

vi 最早由 Bill Joy 于 1976 年发布, Vim 只是 vi 众多的衍生版之一。其中一些衍生版的特性和 vi 非常接近,而另一些则新增了许多新特性, Vim 就属于后者。接下来将会介绍一些比较著名的 vi 衍生版,并简要描述每个衍生版的特点。

9

1.2.1 vi

vi 是 Vim 家族的原始祖先,由 Bill Joy 在 1976 年开发,系统平台是 BSD (Berkeley Software Distribution) 的一个早期版本。vi 是当时最流行的编辑器 ex 的扩展版本。而 ex 则是 Unix 编辑器 ed 的扩展版本。“vi”的意思是 visual in ex,顾名思义,vi 仅仅是一个命令,命令的作用是以可视化模式启动 ex 编辑器。

vi 是最早引入模式 (modality) 概念的编辑器之一。模式指的是在处理不同的任务时,编辑器可以处于不同的工作模式——有的模式用来编辑文本,有的模式用来选择文本,还有一些模式用于执行命令。

模式是 vi 的主要特性之一,这个特性使得热爱 vi 的人更加热爱它,但也会让讨厌 vi 的人更加讨厌。

自从第一个版本发布之后,vi 就没有发生过比较大的变化,但这并不妨碍它成为 Unix 社区最流行的编辑器之一,这主要是因为 SUS (Single Unix Specification) 把 vi 列为 Unix 系统的必备软件之一——只有符合 SUS 的系统才能称之为 Unix 系统。

1.2.2 STEVIE

1987 年, Tim Thompson 获得了他的第一台 Atari ST (Sixteen/Thirty-two),但这个平台还没有一款优秀的编辑器可供使用,于是,他决定把 vi 移植到这个平台中。1987 年 6 月,他发布了一款编辑器,所使用的授权类似于后来的开源协议。他在新闻组中发布了这款编辑器,并取名为 STEVIE——意为 ST Editor for VI Enthusiasts。

这款编辑器非常简单,只支持 vi 的一小部分功能,但是它提供了一个对 vi 用户来说非常熟悉的环境,这可以帮助他们在 ST 平台上继续高效地工作。

发布之后, Tim Thompson 停止了 STEVIE 的开发工作,但是 Tony Andrews 马上接手了过来,并且在一年内,就把它移植到了 Unix 和 OS/2 系统中。在这过程中,越来越多的特性被加了进来,但是到了 1990 年,开发工作又停止了。

虽然 STEVIE 只生存了几年,但是 Tim 和 Tony 把源代码放到了新闻组上,任何人都可以免费地浏览和下载,正因为如此,后来的许多 vi 衍生版都或多或少从这些代码中得到启发,或以它们为基础再加以开发。

10

1.2.3 Elvis

STEVIE 是比较流行的编辑器之一,但是它还有许多问题,限制也比较多。当时还在使用 Minix 的 Steve Kirkendall 注意到了 STEVIE 的一个缺点:在编辑文件时,它会把整个文件读取到内存中,对 Minix 来说这并不是一个很明智的做法。于是 Steve 决定修改 STEVIE,让编辑器把一个文件当做缓冲区使用,而不是在内存中编辑,这产生了 Elvis v1.0。

虽然和 vi 比起来,Elvis 已经得到了很大的改善,但是它仍然受到同样的限制——行的最大长度,以及单文件缓冲区。

为了完全摆脱这些限制, Steve Kirkendall 决定重写 Elvis, 这产生了 Elvis v2, 当前可用的版本是 v2.2. 在 Elvis 的第 2 版中, Steve 添加了许多 vi 原来没有的特性, 其中比较重要的有:

- 语法高亮
- 多窗口支持
- 网络支持 (HTTP 和 FTP)
- 简单的 GUI 前端

Elvis 现在已经停止了开发, 但仍然被广泛地使用, Elvis 支持的平台包括 Unix, MS Windows (控制台程序, 或带有 GUI 的 WinElvis), 以及 OS/2.



关于 Elvis 的最新版请访问 <http://elvis.the-little-red-haired-girl.org/>.

1.2.4 nvi

nvi, 全称 new vi, 是 AT&T 与加州大学伯克利分校 Computer Science Research Group (CSRG) 授权争论的结果. vi 使用了 ed 的源代码, 而 ed 使用了 AT&T System V Unix 授权, 所以 CSRG 无法使用 BSD 授权发布 vi, 于是他们决定发布 vi 的替代版本.

11

新 vi 的开发人员是 Keith Bostic. vi 克隆了已经免费的 Elvis, 但是 Keith 希望新的编辑器和原始的 vi 尽量保持接近, 于是他采用了 Elvis 的代码, 并开发出一个和 vi 完全兼容的衍生版 — nvi. 在原始 vi 的功能集中, 只有 打开模式 (Open Mode) 和 lisp 编辑 (lisp edit) 被剔除出去.

随着 4.4BSD 的发布, nvi 完全替代了 vi 编辑器, 而且再次使用了完全免费的授权.

如今, 大多数基于 BSD 的操作系统都使用了 nvi 作为它们的默认 vi 编辑器, 包括 NetBSD, FreeBSD, 还有 OpenBSD, 并且功能也越来越丰富.

和原始的 vi 相比, nvi 包含的新特性主要有:

- 多个编辑缓冲区
- 不限次数的撤消
- 扩展的正则表达式
- 支持 CScope
- 简单的脚本支持, 脚本语言可以是 Perl 或 Tcl/Tk

Keith Bostic 现在依然在维护 nvi, 但是开发量已经少了很多.



可以到 <http://www.bostic.com/vi/> 获取最新版的 nvi.

1.2.5 Vim

Vim 编辑器是 vi 家族的天之骄子. 自从 Bram Moolenaar 在 1991 年发布了 Vim 的第 1 版之后, 这个编辑器已经成长为世界上功能最丰富的编辑器之一. Vim 的第 1 版和 Elvis 一样, 都是基于 STEVIE 的源代码. 然而, Bram 发布的 Vim 只支持 Amiga 平台, 在当时, Amiga 是家庭计算机中最流行的平台之一. 那时候 Vim 还只是 Vi IMitation 的缩写形式, 事实上也的确是这样, 因为当时仅仅希望 Vim 能尽量模拟 vi 的功能.

12

可是就在一年之后, 到了 1992 年, Bram 把 Vim 移植到了 Unix 平台, 这样做的结果是 Vim 已经不仅仅是 vi 的衍生版, 而是变成了一个强有力的竞争对手. Vim 的开发工作非常迅速, 仅用了很短的时间, Vim 就拥有了许多原始 vi 不支持的特性. 也正是由于这个原因, Vim 的全称变成了 Vi IMproved.

两年之内, Vim 就拥有了一大堆 vi 用户朝思暮想的功能, 这也使得越来越多的 vi 用户转而投向 Vim 的怀抱.

1998 年, Vim 的第 5 代版本发布, 这个版本包含了如今使用得最广泛的一个特性: 脚本编程.

现在, 用户可以根据自己的需要, 通过编写脚本来扩展 Vim 的功能. 这是一个非常大的进步, 因为在这以前, 如果想要为编辑器添加一个小功能, 通常需要用比较低级的语言编码, 然后再重新编译 — 整个过程非常麻烦.

最近十年, Vim 添加了许多新特性, 和其他编辑器, 以及 vi 的其他衍生版比起来, 其中的许多特性显得非常与众不同.

因为完整的列表比较长, 所以在这里仅列出 Vim 最显著的一些特性:

- 在多个缓冲区, 窗口, 和标签页中同时编辑多个文件
- 高级脚本语言
- 支持 Perl 与 Python
- 支持超过 200 种语言的语法高亮
- 带有分支的, 不限次数的撤消/重做
- 根据上下文补全单词和函数
- 借助正则表达式, 支持高级的模式匹配
- 与多种编译器, 解释器, 以及调试器的紧密结合
- 可在线免费获取超过 1500 份 Vim 脚本程序

13

Vim 所支持的平台非常广泛, 比如所有的 Unix 系统, Linux, MS Dos, MS Windows, AmigaOS, Atari MiNT, OS/2, OS/390, MacOS, OpenVMS, RISC OS, 以及 QNX.

1.2.6 Vile

Vile 也许是所有衍生版中最不像 vi 的一个 — 甚至有人认为它根本就不是 vi 的衍生版. Vile 试图把世界上最好的两个编辑器集成在一起 — vi 与 Emacs.

这也解释了名字 Vile 的来源 — VI Like Emacs.

Paul Fox 在 1990 年的夏季启动了 Vile 的开发计划, 其源代码基于公开的 MicroEmacs 编辑器, 在后面的日子里, Paul 为它添加了模式与其他的类 vi 特性.

MicroEmacs 并没有具备 Emacs 编辑器的全部特性, 但是它支持长文本行, 以及在多个窗口中同时编辑多个文件. 这些特性 vi 并不具备, 但是许多程序员却很需要它们.

为了把 **MicroEmacs** 改造成类 **vi** 编辑器, 开发人员做了大量的工作. 在这过程中陆续有几位新成员加入到开发团队中, 其中就包括 **Thomas E. Dickey**, 他在 1992 参与进来, 他为 **Vile** 添加了许多新特性, 也修复了很多问题.

1994 年, **Kevin Buettner** 参与了进来, 着手开发 **Vile** 的 GUI 版本 — **xvile**, **xvile** 支持当时最常用的一些图形部件, 比如 **Athena**, **OpenLook**, **Motif**, 以及 **Xt Toolkit**.

如今 **Thomas** 是 **Vile** 的主要维护人员, 开发工作也由他掌控. 但是他花在开发上的时间十分有限, 在大多数情况下只是做一些问题修复工作.

Vi 与 **Vile** 在工作方式上不太一样, 而且只有一小部分的 **vi** 特性被保留到了 **Vile** 中. **Vile** 的主要特性包括:

- 编辑模式 — 为每一种文件类型配备一个模式
- 为了支持宏而添加的 **Vile** 过程语言
- **Perl** 支持 (实验性的)
- 根据用户的需要, 把命名函数绑定到按键上

Vile 支持的平台包括 **Unix**, **Linux**, **BeOS**, **OS/2**, **VMS**, 和 **MS Windows**, 每一种平台都同时拥有控制台版本和 GUI 版本.

14



关于最新版的 **vile** 可以访问 <http://invisible-island.net/vile/vile.html>.

1.2.7 兼容性

虽然所有的 **vi** 衍生版在某种程度上都试图表现得像 **vi** 一样, 但是其中大多数其实在朝着完全不同的方向发展. 这就意味着, 虽然它们都支持一些共同的功能, 比如语法高亮, 但是这并不要求一定要用同样的方法来实现. 因此, 如果有一个 **Vim** 的语法文件, 那它就不能被 **Elvis** 所使用.

即使是那些起源于 **vi** 的特性, 其实现方法也不一定相同. 有些衍生版所实现的特性不如其他衍生版准确. 也许特性背后的思想都是相同的, 但是使用它们的结果却有可能完全不同.

在下面这张表中, 笔者尽量精确地给出上面所提到的衍生版和 **vi** 的相似度百分比 (兼容性的最低值用 0% 表示, 100% 表示完全兼容). 它们之间的比较说明了开发人员为了尽量精确地实现 **vi** 特性所付出的努力.

衍生版	vi 兼容性	说明
STEVIE	10%	相同的特性集只有很小的一部分
Vile	10%	相同的部分只包括常规概念, 比如模式
Elvis	80%	大量的特性集相同, 但是还有一些特性表现地很不相同
Nvi	95%	接近完全兼容, 但是还是有一小部分特性表现地不一样
Vim	99%	在“兼容模式”下, 几乎所有的特性都兼容

表格只考虑了衍生版和 **vi** 共同拥有的特性, 也就是说虽然 **Vim** 拥有许多 **vi** 并不支持的特性, 但是在共同支持的特性上, **Vim** 仍然表现得和 **vi** 非常类似. 除此之外, **Vim** 几乎实现了 **vi** 的所有特性, 有一些特性, **Bram Moolenaar** 把它们当作 **vi** 的缺陷, 所以在 **Vim** 中用了不同的方法来实现. 注意, 为了能让 **Vim** 达到 99% 的 **vi** 兼容性, 你必须执行下列命令, 把它设置成兼容模式:

```
:set compatible
```

15



如果读者想要了解更多的关于 vi 与 Vim 之间的差异, 在 Vim 中输入命令:

```
:help vi-differences
```

另一个比较有趣的地方是, 虽然 STEVIE 非常精确地实现了 vi 的一部分功能集, 但由于比例过低, 所以人们并没有把它当作 vi 的近亲.

1.3 博爱的 Vim

Vim 的开发人员 — Bram Moolenaar — 选择了一种称为慈善授权的许可证来发布 Vim. 该许可证允许用户不受限制地复制 Vim, 但是它鼓励大家向慈善团体捐赠.

如果你想要了解更多的关于捐赠的信息, 在打开 Vim 后, 执行

```
:help uganda
```

除此之外, 用户还可以访问 <http://www.vim.org/sponsor/>, 获取更多的关于如何赞助 Vim 项目的资料.

如果用户是一名 Vim 赞助人, 就有权利投票把哪些新特性加入到 Vim 中. 所以, 赞助不仅可以帮助他人, 还可以影响 Vim 的演变.

1.4 公共术语

有些单词在不同的语境中会拥有不同的涵义, 因此, 先在这里解释一些本书将会用到的重要术语:

- 黑客 (Hacker): 技术迷, 喜欢钻研与技术相关的事物, 并对它们进行优化, 以满足自己或他人的需求.
- Hacking: 黑客对软件或硬件进行探索和优化的过程. 因此 “Hacking Vim” 可以理解成为了满足黑客 (用户) 的需求, 对 Vim 编辑器进行优化的过程.
- 脚本 (Script): 由一行或多行代码组成的文本程序, 可以通过解释器来运行. 一个脚本程序可以用多种不同的编程语言来编写, 但是在运行时必须选择一种与编程语言相匹配的解释器.
- 解释器 (Interpreter): 一种程序, 它可以从脚本中读取行, 然后一行接一行地加以执行. 解释器可以被构建到另一个程序中 (比如 Vim).

16

1.5 小结

这一章简单地介绍了 Vim, 以及本书所关注的内容. Vim 只是历史悠久的 Unix vi 编辑器的众多衍生版之一, 为了让读者对 vi 家族有一个感性的认识, 我们介绍了几种比较著名的 vi 衍生版. 它们的历史, 以及和 vi 之间的联系, 只进行了简单的介绍, 另外我们还了解到, 虽然这些衍生版都试图向 vi 靠拢, 但它们之间并非完全兼容.

接着简单地介绍了一些关于慈善授权的概念, 以及作者如何利用 Vim 来筹集善款, 从而帮助乌干达的儿童. 最后, 介绍了书中比较重要的几个术语, 并解释了本书书名 “Hacking Vim” 的意义.

现在, 读者可以继续往下阅读, 学习如何定制 Vim 编辑器.

第二章 定制 Vim

17

如果读者经常使用计算机编辑文件, 那么应该会马上意识到拥有一款优秀的编辑器是多么的重要. 一款优秀的编辑器能够成为你的好友, 帮助你高效地完成日常工作. 但是什么样的编辑器才能称之为优秀呢?

通过观察不同的编辑器可以发现, 开发人员努力往编辑器身上加入他们认为用户需要的特性, 籍此希望编辑器越来越好用, 成为顶尖中的顶尖. 但是还有一些编辑器勇于承认自己的不足, 它们只希望能够尽量简单, 对用户友好, 能够更快速地启动.

有了 Vim 编辑器, 没有人能够决定谁才是用户的最佳选择. 相反, 用户可以根据自己的需要, 对 Vim 进行大范围的修改, 使它更符合自己的口味: 力量掌握在用户手中, 而不是编辑器的开发人员.

有些设置和 Vim 的布局有关 (比如配色和菜单), 还有些设置可以影响用户使用 Vim 的方式, 比如按键绑定 — 把某个组合键映射到特定的操作上.

本章将介绍一些 Vim 配置方法, 它们可以帮助用户把 Vim 配置成最适合自己的编辑器.

这些方法主要针对下列配置操作:

- 修改字体
- 修改配色方案
- 个性化的高亮
- 更丰富的状态行
- 切换菜单与工具条
- 添加自己的菜单与工具条按钮
- 个性化的工作区

18

其中有些配置可用多种方法来实现, 读者可以根据自己的喜好来选择.

在正式使用 Vim 之前, 还需要知道一些和安装相关的事情, 比如 Vim 配置文件的存放位置.

2.1 配置文件的存放位置

使用 Vim 时, 用户必须了解它的一系列配置文件. 这些配置文件的存放位置取决于 Vim 的安装目录, 以及所使用的操作系统.

通常来说, 用户只需要知道如何找到以下 3 个配置文件即可:

1. vimrc

2. gvimrc

3. exrc

vimrc 是 Vim 的主要配置文件, 它有两个版本 — 全局的和个人的。

全局的 vimrc 存放在 Vim 系统文件的安装目录, 为了找到该目录, 需要启动 vim, 并在普通模式下执行

```
:echo $VIM
```

输出的内容有可能是

- Linux: /usr/share/vim/vimrc
- Windows: c:\program files\vim\vimrc

个人的 vimrc 在用户的家目录, 家目录的位置取决于用户所使用的操作系统. Vim 主要是为 Unix 系统开发的, 所以个人的 vimrc 会在文件名前加一个句点符号, 从而隐藏起来. 在 Unix 中, 以句点开始的文件名可以把文件隐藏起来, 但在 Microsoft Windows 中不起作用, 作为替代, 在这些系统中 vimrc 的文件名以下划线开始. 比如

- Linux: /home/kim/.vimrc
- Windows: c:\Documents and Settings\kim_vimrc

19

个人的 vimrc 配置会覆盖全局的 vimrc 配置效果, 因此, 即使用户没有权限修改 (甚至访问) 全局 vimrc, 也可以通过修改个人的 vimrc 来得到自己想要的配置效果.

如果想要得到用户的家目录路径, 只需要在 Vim 的普通模式下执行命令:

```
:echo $HOME
```

获取个人 vimrc 路径的另一个方法是在普通模式下执行:

```
:echo $MYVIMRC
```

vimrc 文件包含 ex (vi 预处理器) 命令, 每行一个, 在 Vim 启动过程中, 会自动从该文件中读取配置命令. 在剩下的内容里, 我们把 Vim 的配置文件都称为 vimrc, 而不考虑它的具体存放路径.

vimrc 可以把其他文件作为配置信息的外部来源, 在 vimrc 中, 可以这样使用 source 命令:

```
source /path/to/external/file
```

这种方法可以使 vimrc 保持简练, 也可以使配置文件更加结构化. (关于如何保持 vimrc 简练的更多方法, 请参考附录 B)

gvimrc 是专门给 Gvim 使用的配置文件, 类似于 vimrc, 个人的 gvimrc 存放目录与 vimrc 相同, 全局的配置同样如此. 比如:

- Linux: /home/kim/.gvimrc 与 /usr/share/vim/gvimrc
- Windows: c:\Documents and Settings\kim_gvimrc 与 c:\Program Files\vim\gvimrc



特定于 GUI 的配置项只有 Gvim 才能使用, 在剩下的内容里, 我们把 Gvim 的配置文件都统一称为 gvimrc.

gvimrc 不能替代 vimrc, 它仅仅是为了给 Vim 的 GUI 版本提供配置信息. 换句话说, 读者没必要把相同的配置信息同时写在 vimrc 与 gvimrc 两个文件中.

文件 exrc 只是为了保持对编辑器 vi/ex 的向后兼容. 它的存放位置与 vimrc 相同 (包括个人与全局), 使用方法也一样, 不过它很少被用到, 除非用户把 Vim 设置成 vi 兼容模式.

20

2.2 更改字体

对于普通版的 Vim 来说, 更改字体通常没什么好做的, 因为 Vim 的字体与终端字体一致. 然而, 对于 Gvim, 用户可以根据自己的需要随意地修改字体.

在 Linux 中, 设置字体的主要命令是 `guifont`, 例如:

```
:set guifont=Courier\ 14
```

可以把 Courier 替换成你喜欢的任意一种字体的名字, 字体大小 14 也可以根据需求进行更改 (字体大小的单位是点 — pt).

在 Windows 中修改字体用:

```
:set guifont=Courier:14
```

如果不确定系统中是否存在某种字体, 用户可以在命令的末尾再添加一种字体, 两种字体间用逗号分开, 例如:

```
:set guifont=Courier\ New\ 12,Arial\ 10
```

如果字体的名字中包含了空格或逗号, 那就必须用反斜杆转义, 例如:

```
:set guifont=Courier\ New\ 12
```

这个命令把字体设置成 **Courier New**, 大小为 12, 但是该设置仅对当前会话有效. 如果希望每次打开 Vim 时都能够自动设置好字体, 那就得把命令添加到用户的 gvimrc 文件中 (添加到配置文件中的命令需要删除 set 左边的 :))



在 Windows, Linux (使用 GTK+), Mac OS, 或 Photon 的 Gvim 中, 如果执行 `:set guifont=*` 就可以打开一个字体选择窗口.

用户可以根据不同的文件类型 (源代码文件, 普通文本文件, 日志文件等) 设置不同的字体. 例如, 假设用户希望如果打开的是一个普通文本文件 (.txt), 就把字体设置为 **Arial**, 大小 12. 完成这个操作只需要在 vimrc 中添加:

```
autocmd BufEnter *.txt set guifont=Arial\ 12
```

每当字体发生变化时, Gvim 的窗口大小就会自行调整. 这就意味着如果使用了较小的字体, 窗口也会变小. 如果用户像之前说的那样, 对不同的文件类型设置了不同的字体, 每当用户切换到具有不同文件类型的缓冲区时, 字体就会发生变化, 于是窗口大小也会作相应的改变.



用户可以到 Vim 帮助系统的 `Help | guifont` 中找到更多的资料.

21

2.3 修改配色方案

当在终端环境下工作时, 用户常常只能面对黑底白字的屏幕, 这样的屏幕不仅无趣, 而且很昏暗, 如果能加点彩色就好了.

默认情况下, Vim 的窗口颜色与打开它的控制台相同, 但 Vim 还提供了更改颜色的能力. 通常情况下, 为了修改颜色, 只需要修改配色方案文件即可, 这些文件通常放在名为 `colors` 的目录中, 而目录 `colors` 位于 Vim 的安装目录.

为了把配色方案修改成某个系统中已有的方案, 执行:

```
:colorscheme schemename
```

在这个命令中, *schemename* 是系统中已安装的某个配色方案的名字. 如果读者不太清楚系统中已经安装了哪些配色方案, 请在输入

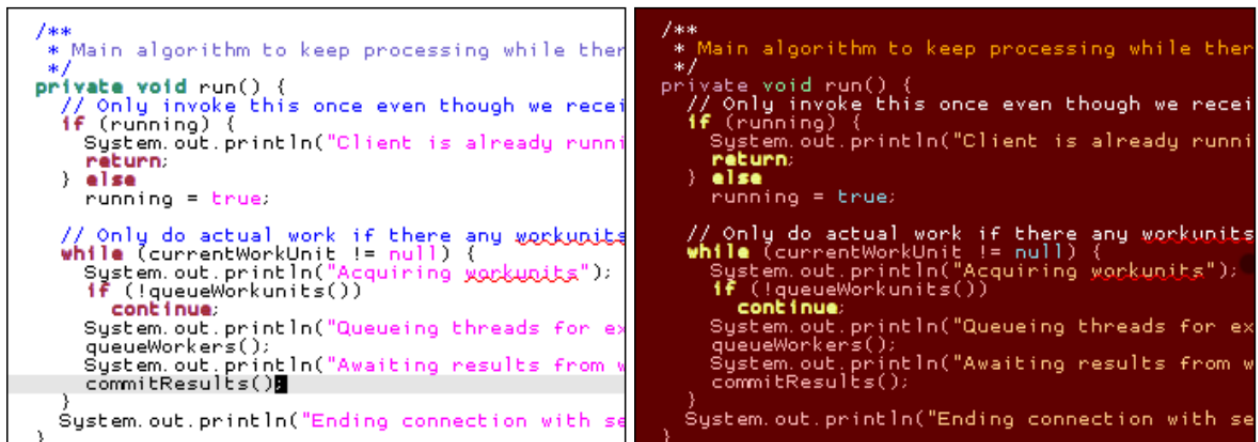
```
:colorscheme
```

之后, 再键入至少一个空格, 然后就可以通过不断地按 *Tab* 键来浏览系统中可用的所有配色方案. 如果看到了一个感兴趣的配色方案, 只需要按下 *Enter* 键就可以马上应用该方案.

除了前景色与背景色, 配色方案还会影响代码高亮, 错误标记, 以及文件中的其他可视化标记.

用户会发现有些配色方案非常类似, 相互之间差别不大, 这是因为很多配色方案都是由用户提供的, 如果用户对某个配色方案不太满意, 他可能会对方案中的某一条配置进行修改, 然后再用另外一个名字重新发布. 为了找到心仪的配色方案, 读者可以把所有的方案都试一遍. 读者可以从现在开始选择一种配色方案, 并查看是否对该方案的所有颜色配置都满意, 在第六章, 我们会回过头来讨论如何改变配色方案的配置, 以满足用户的所有特殊需求.

22



2.3.1 个性化高亮

在 Vim 中, 与高亮相关的最关键的技术是 匹配 (matching).

通过匹配, Vim 几乎可以标记字母, 单词, 数值, 语句, 和文本行的任意一种组合. 用户甚至可以决定如何标记它们 (例如用红色标记错误, 用绿色标记重点内容).

匹配通过下面这个命令完成:

```
:match group /pattern/
```

命令带有两个参数, 第一个参数是高亮文本时使用的色彩组的名字.

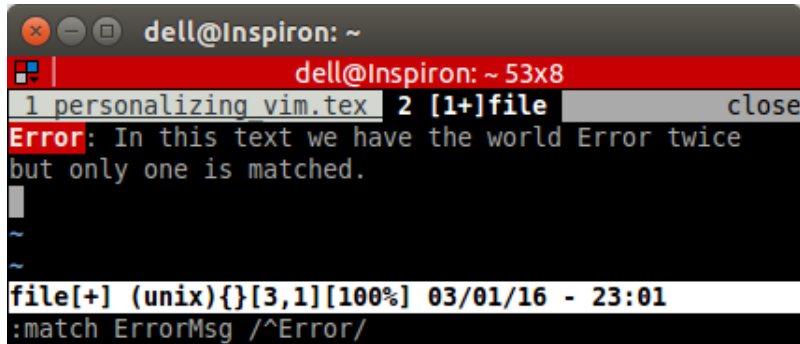


配色方案影响的是全部的颜色设置, 而色彩组只是背景色 (或前景色) 的一个比较小的组合, 用户可以将它应用到某个方面, 比如匹配. 当 Vim 启动时, 根据所选择的配色方案, 大量的色彩组被设置成某个默认值.

为了查看所有的色彩组, 执行 `:so $VIMRUNTIME/syntax/hitest.vim`.

第 2 个参数是待匹配的模式. 模式是一个正则表达式, 可以很简单, 也可以极其复杂 — 取决于用户想要匹配什么样的内容. 匹配命令的一个比较简单的例子是:

```
:match ErrorMsg /^Error/
```



这个命令查找位于行的开始位置的单词 *Error* (带有脱字符 `^`), 如果找到了一个匹配, 就把匹配的文本用色彩组 *ErrorMsg* 的颜色标记起来 (通常是红底白字).

如果读者不太喜欢已有的色彩组, 也可以自己定义一个, 定义色彩组的命令是:

```
:highlight MyGroup ctermbg=red guibg=red gctermfg=yellow
      guifg=yellow term=bold
```

这个命令创建了一个名为 *MyGroup* 的色彩组, 红底黄字, 在控制台 (Vim) 和 GUI (Gvim) 环境下都是如此. 用户可以根据自己的喜好, 修改下列选项:

<code>ctermbg</code>	控制台环境下的背景色
<code>guibg</code>	Gvim 环境下的背景色
<code>ctermfg</code>	控制台环境下的文本颜色
<code>guifg</code>	Gvim 环境下的文本颜色
<code>gui</code>	Gvim 环境下的字体格式
<code>term</code>	控制台环境下的字体格式 (比如粗体: bold)

如果使用了已有的色彩组的名字, 那么在接下来的会话中如果用到了该色彩组, 所使用的将会是修改后的效果.

使用匹配命令时, 给定的模式会一直匹配下去, 直到执行一个新的匹配, 或者执行下列命令:

```
:match NONE
```

匹配命令一次只能匹配一个模式, 但是 Vim 另外提供了 2 个命令用于一次匹配至多 3 个模式. 命令很容易记忆:

```
:2match
:3match
```

读者也许想知道这些匹配命令应该在什么样的情况下使用, 因为在平时的话这些命令没什么大用处. 这里有一些例子, 它们展示出了匹配的强大之处.

23

24

示例 1: 用彩色标记某列后面的文字

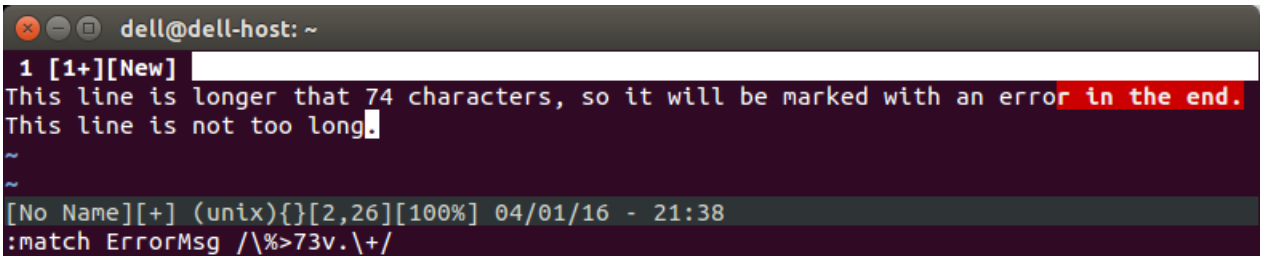
在写邮件时, 一条比较常见的规则是一行的长度不能超过 74 个字符 (在某些古老的编程语言中, 同样存在这样的规则). 在这种情况下, 当用户在一行内写出的字符数超过某个上限时, 如果 Vim 能够发出提醒那就再好不过了.

下面这个命令就可以完成上面提到的功能:

```
:match ErrorMsg /\%>73v.\+ /
```

执行该命令后, 一行内的第 73 个字符之后的那些字符都会被标记成错误. 匹配命令中含有一个正则表达式, 这个表达式可以拆成:

\%>	匹配该列之后的内容, 列号紧跟在尖括号的右边
73	列号
v	只能工作在可见的列上面
.\+	匹配一个或多个任意的字符



示例 2: 标记代码中未被用作缩进的制表符

编码时, 一条很重要的规则是制表符只被用作缩进代码, 其他地方则不允许使用制表符. 然而, 对某些人来说常常会忘记该规则. 现在, 只需要一条简单的匹配命令, 就可以时刻提醒着程序员.

下面这条命令把所有的, 不在行的开始位置上的制表符用错误信息的颜色标记出来:

```
:match ErrorMsg /[^\t]\zs\t\+ /
```

执行完该命令后, 程序员就可以时刻知道编码是否违反了规则 — 把制表符用在了代码内部. 把命令分解开来看, 它由下列这几个部分组成:

[^	字符组的开始标记, 组中的字符将不会被匹配到
\t	制表符
]	字符组的结束标记
\zs	一个宽度为 0 的匹配, 它把“匹配”置于一行的开始, 并忽略任意的空格
\t\+	匹配一个或多个制表符

- 命令行缓冲区 (输入命令的地方)
- 状态行

在默认的配置中, Vim 的状态行非常简单, 信息量也比较少. 状态行的右边显示的是光标当前所在位置的行号与列号, 左边则是当前打开着的文件名 (如果有的话).

当执行 Vim 命令时, 状态行就会消失, 取而代之的是命令行缓冲区. 如果所执行的命令带有输出信息, 那么这些信息就会出现在状态行的右侧. 对于简单的文本编辑来说, 这样的状态行已经足够用了. 但是, 如果用户需要天天使用 Vim 来处理不同格式的文件, 那么一条信息量丰富的状态行将会提供很大的帮助.

27

这一节将会通过几个例子, 说明如何利用简单的办法来配置状态行, 使它具有更丰富的信息.

配置状态行的命令具有形式:

```
:set statusline format
```

命令中的 *format* 表示一个格式字符串 (就像 C 程序的 printf), 它描述了状态行的显示格式.

键入 :help 'statusline', 打开 Vim 的帮助系统, 读者将会看到状态行可以包含相当多的信息, 其中有些信息在日常工作中非常有用.

笔者的状态行信息包括:

- 正在编辑的文件名
- 正在编辑的文件的格式 (比如 Dos, Unix)
- 文件类型
- 光标所在位置的字符的 ASCII 码值与十六进制值
- 光标所在的行号与列号
- 文件的长度 (以行为单位)

下面的命令可以让读者的 Vim 状态行包含上面提到的所有信息:

```
:set statusline=%F%m%r%h%w\ [FORMAT=%{&ff}]\ [TYPE=%Y]\ [ASCII=\%03.3b]\
[HEX=\%02.2B]\ [POS=%04l,%04v]\ [%p%\]\ [LEN=%L]
```

我在每一种信息的左右两边都加上了中括号, 这样做就更容易区分彼此. 加中括号只是为了增加一些视觉效果, 如果不喜欢的话可以直接删掉.

```
1 personalizing_vim.tex
\begin{vimcmd}
:set statusline=%F%m%r%h%w\ [FORMAT=%{&ff}]\ [TYPE=%Y]\ [ASCII=\%03.3b]\ [HEX=\%02.2B]\ [POS
=%04l,%04v]\ [%p%\]\ [LEN=%L]
\end{vimcmd}
<izing_vim.tex [FORMAT=unix] [TYPE=TEX] [ASCII=000] [HEX=00] [POS=0406,0122] [99%] [LEN=407]
-- INSERT --
```

28

然而, 如果在安装 Vim 后没有做过多的配置, 即使执行了上面的状态行命令, 用户也看不到状态行有什么变化. 这是因为默认情况下 Vim 根本就不会显示状态行, 它只会显示命令行缓冲区及其一小部分信息. 为了让 Vim 显示出真正的状态行, 用户必须把下面这条命令添加到 vimrc 文件. 这条命令确保状态行始终显示在倒数第 2 行的位置上:

```
:set laststatus=2
```

执行完该命令后, 读者就会发现命令行缓冲区有了自己的位置: 窗口的最后一行. 通过这条命令就可以保证状态行始终拥有自己的显示位置, 而用户也就能一直看到文件的相关信息. 当然, 状态行会占据编辑区的一部分空间, 但是是否显示状态行由用户来决定. 下面这条命令关闭状态行, 直到当前会话结束:

```
:set laststatus=0
```

2.5 切换菜单与工具条

如果用户是在控制台中使用 Vim, 就会逐渐习惯没有菜单与工具条的窗口. 但是如果用的是 Gvim, 就会发现窗口界面提供了菜单与工具条.

许多人认为应该把额外的空间分配给更重要的文本使用, 而不是菜单与工具条. 如果读者也是这么认为的, 那么读者可能希望在使用 Gvim 时, 移除菜单与工具条. 然而, 某些脚本在菜单中增加许多有用的功能, 而这些功能也只能通过菜单来使用. 解决办法是根据实际需要, 随时切换菜单与工具条.

下面的代码把 Gvim 的组合键 **Ctrl+F2** 映射到菜单与工具条的切换操作上, 如果读者喜欢这个功能的话, 可以把代码写到 vimrc 中.

```
map <silent> <C-F2> :if &guioptions =~# 'T' <Bar>
                        \set guioptions-=T <Bar>
                        \set guioptions-=m <bar>
                    \else <Bar>
                        \set guioptions+=T <Bar>
                        \set guioptions+=m <Bar>
                    \endif<CR>
```

现在, 无论何时, 只要用户想关闭菜单与工具条, 好把空间留给文本, 只需要按下 **Ctrl+F2** 即可, 如果想它们了, 再按一下 **Ctrl+F2**, 它们就回来了. 如果用户希望一直隐藏菜单或工具条, 只需要在 vimrc 中添加下面两个命令中的一个即可.

为了完全移除菜单, 添加:

```
:set guioptions-=m
```

为了完全移除工具条, 添加:

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



GUI 的其他设置可以通过命令 `set guioptions` 完成, 更多的信息可以参考: `help 'guioptions'`.

29

2.6 添加自定义菜单与工具条按钮

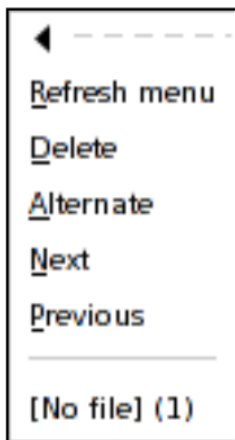
如果使用的是 Gvim, 那么用户可以把经常用到的功能加到菜单中. 也许用户并不总是需要通过菜单来使用某项功能, 但是如果忘记了具体的操作步骤, 还可以通过菜单来完成. 如果用户确实需要快速地执行某个功能, 还可以把它直接加入到 Gvim 的工具条中.

这一节讨论如何在 Gvim 中制作菜单, 以及如何往工具条中加入额外的按钮. 先从构造菜单开始.

添加菜单

简单来说, 为了构造一个菜单, 只需要为菜单中的每一个菜单项执行对应的命令即可. 只要遵循良好的命名规则, 用户就可以得到一个使用方便, 功能齐全的菜单. 先从小例子开始, 假设用户想新增一个菜单, 比如缓冲区菜单.

30



需要用到的命令是:

```
:menu menupath command
```

这个命令的工作方式类似于命令 `map`, 后者把命令映射到一个组合键, 而前者是把命令映射到一个菜单项.

命令带有两个参数. 第 1 个参数是菜单项在菜单中的实际路径, 第 2 个参数是菜单项被点击时所执行的命令. 举个例子, 假设用户想要在菜单项 `Tabs` 中, 添加一个新菜单项 `Next`, 执行:

```
:menu Tabs.Next <ESC>:tabnext<cr>
```

之后, 用户就有了一个新菜单项 `Tabs`, 它只有一个子菜单项 `Next`. 现在, 如果用户点击菜单项 `Next`, 就会执行:

```
:tabnext
```

命令前面的 `<ESC>` 为的是让 Vim 切换到普通模式, 后面的 `<cr>` 是真正开始执行命令的地方. 如果漏掉了 `<ESC>`, 那么命令就无法正常地工作. 另外一种办法是根据当前的模式, 在命令前添加特定的菜单项, 为此, Vim 提供了几个 `:menu` 的替代命令:

- `:nmenu` 用于 普通 (Normal) 模式
- `:imenu` 用于 插入 (Insert) 模式, 前置 `^O`
- `:vmenu` 用于 可视 (Visual) 模式, 前置 `^C`, 后置 `^\^G`
- `:cmenu` 用于 命令行 (Command-line) 模式, 前置 `^C`, 后置 `^\^G`
- `:omenu` 用于 操作挂起 (OP-pending) 模式, 前置 `^C`, 后置 `^\^G`

前置部分 (^O 与 ^C) 把 Vim 切换到普通模式。

^O (Ctrl+O) 专门用于插入模式, 因为它会在执行完命令之后让编辑器重新回到插入模式。

^\\^G (Ctrl+\\, Ctrl+G) 用于这样一种特殊情况: 全局插入模式被设置为真, 并且 Vim 把插入模式作为默认模式 (此时 Vim 是无模式的), 此时, 它们就可以把模式还原成原来的样子。

没必要为每一种模式创建一个同样的菜单项, 这时候只需要使用下面这个命令即可:



```
:amenu menu-path command
```

这个命令可以根据当前的模式, 设置正确的前置命令与后置命令。

现在开始创建新菜单 **Tabs**, 并添加几个新菜单项与功能。执行了下面这些命令后, 新菜单就有点像 **Buffers** 菜单了:

```
:amenu Tabs.&Delete :confirm tabclose<cr>
:amenu Tabs.&Alternate :confirm tabn #<cr>
:amenu <silent> Tabs.&Next :tabnext<cr>
:amenu <silent> Tabs.&Previous :tabprevious<cr>
```

细心的读者会发现这些命令之中又多了点新东西。

第一个是 <silent>, 它可以阻止命令执行过程中, 命令的内容被回显到命令行缓冲区。当然, 这个功能只是装饰性的, 相比之下, 菜单路径中的 & 就实用多了。把字符 & 加在菜单路径的最后一个部分的某个字母之前, 就可以为该菜单项定义一个快捷键。有了快捷键, 定位并执行菜单项就可以更加方便。



假设用户想通过执行菜单项 **Tabs | Next** 来切换到下一个标签页, 现在, 只需要按下 **Alt + T + N** 即可。Alt + T 打开 **Tabs**, 再按 **N** (之所以是 **N**, 是因为 & 放在 **Next** 的 **N** 的前面) 来执行 **Next**。如果有其他的菜单项使用了相同的快捷键字母, 只需要一直按 **Alt** 就可以遍历所有的菜单项。

如果用户希望下拉菜单的各项之间用一条横线分开, 可以使用 **SEP** (针对菜单项) 或 **:"** (针对命令):



```
:amenu Tabs.-SEP-
```

用户创建的菜单仅对当前会话有效, 如果想要让菜单在下次启动 Vim 时仍然存在, 可以把菜单的创建命令写到 **vimrc** 中 (写到 **vimrc** 的命令不需要前面的冒号 :).

现在已经有有了一个简单的 **Tabs** 菜单, 看起来有点像菜单 **Buffers**. 不过它还不具备 **Buffers** 菜单所拥有的列出活动缓冲区的功能. 缓冲区可以对用户隐藏起来, 而标签页肯定是可见的, 如果读者知道这点, 就会明白这个功能对标签页来说没什么用处. 换句话说, 用户看见的标签页就是当前编辑器中所有的标签页, 而且它们的名字都显示在标签的标题栏中.

一个 **Personal** 菜单可以用来完成许多有趣的事情. 如果用户需要处理多种类型的文件, 甚至可以启动时为某个特定的文件类型创建菜单, 或者是在同一个菜单下, 为不同的文件类型搭配不同的子菜单.

只需要遵循菜单路径的命名规范就可以创建出子菜单. 于是, 如果用户想要的菜单是 **Tabs | Navigation | Next**, 只需要用菜单路径 `Tabs.Navigation.&Next` 来添加子菜单项 **Next**.

添加工具条图标

既然已经知道了如何创建菜单, 那么给工具条添加自定义的图标就容易多了. 事实上, **Vim** 把工具条看作是带有特殊名字的菜单. 因此, 给工具条添加图标就像是给菜单添加菜单项.

对于工具菜单, 为了给它添加一个子项目, 需要使用以 `ToolBar` 开始的菜单路径. 为了给工具条添加一个用于执行命令 `:buffers` (列出打开的缓冲区) 的子项目, 用户需要执行命令:

```
:amenu icon=/path/to/icon/myicon.png ToolBar.Bufferlist :buffers<cr>
```

当然, 用户必须把图标文件放在某个目录中, 并在命令中给出这个目录.

图标文件的存放目录通过参数 `icon` 传递给命令 `amenu`. 如果只给出了文件名, 而没有给出路径, 那么 **Vim** 就会在运行时路径的 `bitmaps/` 目录下搜索图标文件 (运行时路径可以通过 `:echo $VIMRUNTIME` 命令获取). 受支持的图标类型依赖于系统.

到现在为止就大功告成了! 执行完命令之后, 用户就可以看到他的图标出现在了工具条中, 而且处在已有图标的最靠右的位置上. 如果点击该图标, **Vim** 就会执行命令 `:buffers`, 显示缓冲区列表.

和菜单一样, 可以通过特定于模式的菜单命令 `imenu`, `vmenu`, `cmenu` 等, 使得工具按钮只会在特定的模式下才会显示出来.



默认情况下, 新增的菜单或按钮图标处在已有的菜单或图标的右边, 可以通过优先级来改变默认行为. 更多的信息请参考 `help menu-priority` 与 `:help sub-menu-priority`.

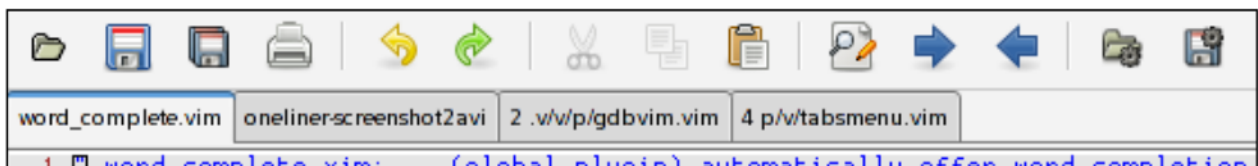
33

2.7 修改标签页

Vim 从 7.0 开始支持标签页. 标签页与其他应用程序中的标签不太一样 — 在 **Vim** 中, 它是组织打开文件的一种方式. 每一个标签都可以包含若干个缓冲区, 甚至多个窗口.

有了标签页之后, 原来的针对所有缓冲区或窗口的命令 (例如 `:bufdo`, `:windo` 和 `:ball`) 将仅限于当前标签的窗口或缓冲区.

一般情况下, 所有的标签页以一种标签列表的形式呈现在窗口的上方 (就在编辑区之上). 每一个标签页都有一个标签, 默认是处于当前活动缓冲区的文件名. 如果一个标签页中同时打开了多个窗口, 那么标签也会显示窗口的数量.



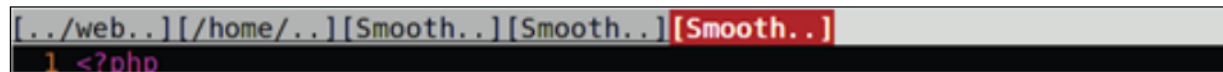
有时候用户希望标签页的标签能提供更多的信息,比如,如果用户需要为每一个项目建一个标签页,要是能根据项目的名称来对标签页进行命名的话,那就方便多了。

34



设置标签名的方法与状态行 (见 2.4 节) 非常类似. 但是在这里设置的是 tabline 的属性, 而不是 statusline:

```
:set tabline tabline-layout
```



如果是 Gvim, 则是

```
:set guitablabel
```

虽然 tabline 的设置方法类似于 statusline, 但是前者会更麻烦一点. 麻烦的主要原因在于用户必须时刻注意标签页是否处于活跃状态. 先从 Vim 的一个小例子开始.

如果标签页过多, 标签就会占据太多的空间, 尤其是当它们包含了当前活跃缓冲区的完整文件名时. 我们希望只显示活跃缓冲区文件名的前 6 个字母, 而且当前活跃的标签名以红底白字的方式呈现, 就像错误信息那样.

下面的 Vim 脚本程序完成的正是这个功能 (关于如何编写 Vim 脚本程序见第六章).

```
function ShortTabLine()
    let ret = ''
    for i in range(tabpagenr('$'))
        " select the color group for highlighting active tab
        if i + 1 == tabpagenr()
            let ret .= '%#errorMsg#'
        else
            let ret .= '%#TabLine#'
        endif

        " find the buffername for the tablabel
        let buflist = tabpagebuflist(i+1)
        let winnr = tabpagewinnr(i+1)
        let buffername = bufname(buflist[winnr - 1])
        let filename = fnamemodify(buffername, ':t')

        " check if there is no name

        if filename == ''
            let filename = 'noname'
        endif

        " only show the first 6 letters of the name and
        " .. if the filename is more than 8 letters long
```

35

```

        if strlen(filename) >=8
            let ret .= '['. filename[0:5]. '...']'
        else
            let ret .= '['. filename. ']'
        endif
    endfor

    " after the last tab fill with TabLineFill and reset tab page #
    let ret .= '%#TabLineFill#%T'
    return ret
endfunction

```

现在, 需要把函数添加到 vimrc, 除此之外, 还要增加一行设置 tabline 的命令, 具体的设置命令是:

```
:set tabline=%!ShortTabLine()
```

命令的执行结果是产生了一个更加紧凑的标签列表, 下面是效果截图:



在 Gvim 中设置 tabline 的方法稍有不同, 不过基本思想是一样的. 然而在 GUI 中, 用户不必考虑活跃标签的颜色, 也不必担心某个标签当前是否处于活跃状态 — 这些都是 GUI 自己需要考虑的东西.

于是, 可以对函数 ShortTabLine() 进行简化 — 只需要设置标签名即可:

```

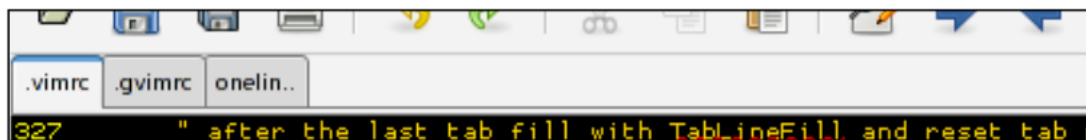
function ShortTabLabel()
    let bufnrlist = tabpagebuflist(v:lnum)
    " show only the first 6 letters of the name + ..
    let label = bufname(bufnrlist[tabpagewinnr(v:lnum) - 1])
    let filename = fnamemodify(label, ':h')
    " only add .. if string is more than 8 letters
    if strlen(filename) >=8
        let ret=filename[0:5]. '...'
    else
        let ret = filename
    endif
    return ret
endfunction

```

再把 guitablabel 的属性设置成函数的输出:

```
:set guitablabel=%{ShortTabLabel() }
```

命令的执行效果是:





如果用户想要完全移除 Gvim 的标签栏, 执行 `:set showtabline=0` (把 `showtabline` 设置成 1 则会重新显示标签栏).

现在, 我们已经拥有了只含有有限信息的标签, 但还是希望能从其他地方得到信息, 为了完成这个功能需要用到一个小技巧 — 使用工具提示.

工具提示的优点是当用户没有激活它们时 (例如, 把鼠标移动到标签页的位置上, 就有可能激活标签页的提示信息), 就不会看到这些提示信息. 使用这种方法就可以做到在不填满整个编辑器的情况下, 仍然可以获取到信息.

为了给一个标签页设置工具提示, 需要使用下面的命令:

```
:set guitabtooltip
```

用户需要把它设置成当鼠标经过标签名时, 希望显示出来的信息.

为了测试, 可以先输入一条简单的提示信息:

```
:set guitabtooltip='my tooltip'
```

现在, 工具提示中就可以显示一条静态消息, 但我们需要更多的信息. 文件的路径部分已经从标签中移走了, 但是有时候仍然需要这个信息. 有了工具提示, 就可以把路径信息显示在工具提示中:

```
:set guitabtooltip=%!bufname($)
```

函数可以根据标签的不同, 显示对应的提示信息. 这里, 我们构造了一个小函数, 用于显示用户会在标签中看到的所有信息, 不过用了一种更加有条理的方式显示出来:

```
function! InfoGuiTooltip()
    "get window count

    let wincount = tabpagewinnr(tabpagenr(), '$')
    let bufferlist=''

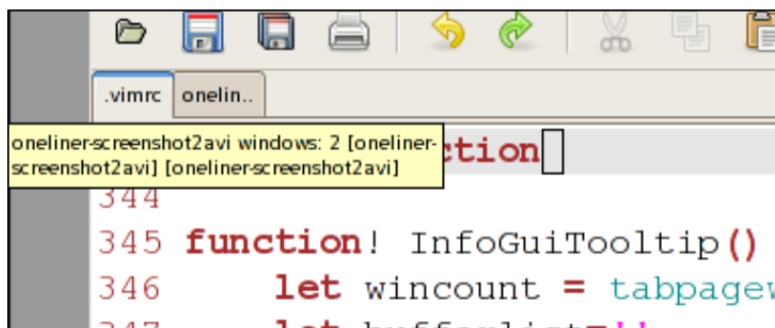
    "get name of active buffers in windows
    for i in tabpagebuflist()
        let bufferlist .= '['.fnamemodify(bufname(i), ':t').'] '
    endfor

    return bufname($).' windows: ' . wincount . ' ' . bufferlist ' '
endfunction
```

然后执行:

```
:set guitabtooltip=%!InfoGuiTooltip()
```

下面这张截图展示了工具提示在 Gvim 中的效果:



用户还可以想到许多其他有趣的用法, 来使用标签与工具提示所提供的功能, 有了前面的示例, 用户就可以自己动手来实现它们了。

2.8 工作区定制

这一节介绍一些专门针对 Vim 编辑区的设置内容, 之所以介绍这些, 是因为当用户使用 Vim 编辑文本或代码时, 它们可以提高用户的工作效率。

2.8.1 为光标添加视觉效果

有时候, 用户正在编辑的文件可能带有多种颜色的语法高亮, 这些颜色会干扰用户追踪光标的位置, 如果可以标记出光标当前所在的行, 那就方便多了。

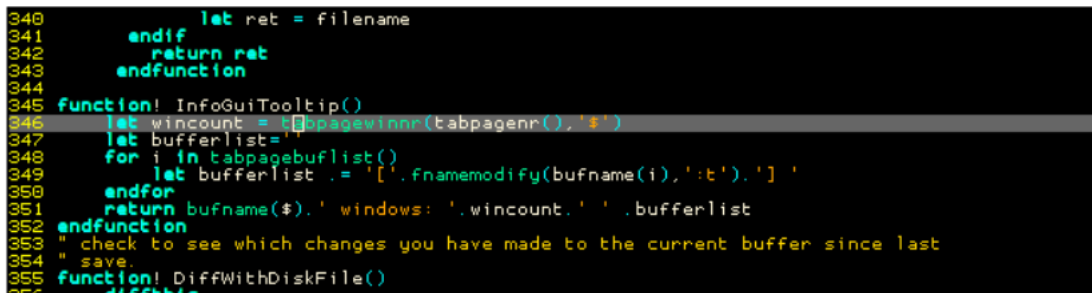
已经有很多人写过 Vim 脚本来解决这个问题, 但是绝大部分都没什么用 (主要是因为太慢了, 当文本过长时, 滚动速度就难以接受)。直到 Vim 版本 7, 这个问题才得以解决, 而且, 它还提供了两种可能的办法来追踪光标的位置。第一种办法是命令 `cursorline`, 它可以标记当前光标所在的行, 比如把当前行的背景色设置成另一种颜色, 而且不会破坏原来的语法高亮。为了打开它, 执行:

```
:set cursorline
```

命令所使用的颜色由色彩组 `CursorLine` 定义, 用户可以它设置成任意一种颜色或风格, 比如:

```
:highlight CursorLine guibg=lightblue ctermbg=lightgray
```

关于色彩组的更多内容请参考 2.3.1 节。



```
340     let ret = filename
341   endif
342   return ret
343 endfunction
344
345 function! InfoGuiTooltip()
346   let wincount = <pagewinnr(tabpagenr(), '$')
347   let bufferlist = ''
348   for i in tabpagebuflist()
349     let bufferlist .= '[' . fnamemodify(bufname(i), ':t') . ']'
350   endfor
351   return bufname($). ' windows: ' . wincount . ' ' . bufferlist
352 endfunction
353 " check to see which changes you have made to the current buffer since last
354 " save.
355 function! DiffWithDiskFile()
```

如果用户正在编辑的文件含有许多对齐的内容 (比如用制表符分隔的数据), 那么很自然地就会用到:

```
:set cursorcolumn
```

这个命令标记光标当前所在的列, 例如, 可以用另外一种颜色来标记。

和 `cursorline` 类似, 用户可以修改 `cursorcolumn` 的标记设置, 它的色彩组是 `CursorColumn`。

如果同时设置了 `cursorline` 与 `cursorcolumn`, 那么光标看起来就像一对十字线, 这样就再也不用怕把它跟丢了。


```

340         let ret = filename
341     endif
342     return ret
343 endfunction
344
345 function! InfoGuiTooltip()
346     let wincount = tabpagewinnr(tabpagenr(), '*')
347     let bufferlist = ''
348     for i in tabpagebuflist()
349         let bufferlist .= '['.fnamemodify(bufname(i), ':t').'] '
350     endfor
351     return bufname($). ' windows: ' . wincount . ' ' . bufferlist
352 endfunction
353 " check to see which changes you have made to the current buffer since last
354 " save.
355 function! DiffWithDiskFile()

```

39



虽然 `cursorline` 与 `cursorcolumn` 是在 Vim 本地实现的, 但是仍然会影响文件的滚动速度。

2.8.2 添加行号

在编译或调试代码时, 错误信息通常会报告错误所在的行号, 为了定位到这一行, 用户当然可以从第 1 行开始, 一行一行地往下数, 不过 Vim 提供了更为直接的办法. 只需要执行 `:XXX` (`XXX` 表示行号), 用户就可以马上到达第 `XXX` 行。

另一种办法是在普通模式 (按下 `Esc` 就可以切换到普通模式) 下, 执行 `XXXgg` 或 `XXXG` (同样, `XXX` 表示行号). 但是, 如果能随时随地地看到行号可能会更加方便, 这时候就可以执行命令:

```
:set number
```

执行完命令后, 编辑器就会在窗口的左边显示每一行的行号. 默认情况下行号占据 4 个空格的宽度, 3 个用于数字, 1 个用于空格. 这意味着行号所占据的宽度总是一样的, 除非总行数超过了 999 行. 如果总行数超过了 999 行, 行号所占据的宽度就会增加一列, 相应地, 文件内容就会右移一列。

当然, 用户可以修改行号占据的列宽, 相应的命令是:

```
:set numberwidth=XXX
```

把 `XXX` 替换成你想要的列宽度。



用户可能希望通过加大列宽度的值, 使得代码与行号之间的间距更大一些, 但是这个配置无法通过 `numberwidth` 实现, 因为行号是右对齐的。

在下面的图片中, 读者可以看到: 当 `numberwidth` 的值增大时, 行号是如何右对齐的。

```

21 When we speak of free sof
22 price. Our General Public
23 have the freedom to distri
24 this service if you wish),
25 if you want it, that you ca
26 in new free programs; and t
27
28 To protect your rights, w
29 anyone to deny you these ri
30 These restrictions translat
~/ontv/COPYING [FORMAT=unix] [TYPE=]

```

40



用户可以通过修改色彩组 `LineNr` 来改变行号及其所在列的风格。

2.8.3 拼写检查

这个道理谁都懂！虽然人们对拼写都很在行，但总会发生拼写错误或打错字的情况。过去，为了进行拼写检查（文本内容已经写在了 Vim 中），用户需要使用某种拼写检查工具，比如 `Aspell` 或 `Ispell`。如果把拼写检查作为最后一项工作，那么修改起来会非常得累人，除非你想过一会儿就检查一遍。

到了 Vim 版本 7，这种麻烦的拼写检查方式终于可以结束了。现在，Vim 内置了一个拼写检查工具，它支持的语言超过 50 种。

拼写检查工具会在用户输入单词的时候检查拼写的正确性，所以如果有错误发生的话，用户可以马上发现。打开拼写检查的命令是

```
:set spell
```

命令使用默认的语言（英语）打开拼写检查，如果想要使用其他语言，只需要把 `spelllang` 设置成该语言的代号即可。比如：

```
:set spelllang=de
```

在这个命令中语言被设置为德语。同一种语言可以用多种形式的格式来表达，比如美式英文可以写成：

- `en_us`
- `us`
- `American`

语言的名字甚至可以是专业名词，比如 `medical`。如果 Vim 无法识别用户输入的语言名字，那么在执行属性设置命令时，它就会高亮显示无法识别的语言名字。



如果把 `spelllang` 设置成某个还没有安装的语言，Vim 就会询问用户是否自动从 Vim 主页上下载对应的语言。

对笔者个人来说，我经常要用 Vim 同时处理不同的语言，而且并不想每次都告诉 Vim 现在用的是什么语言。

对此 Vim 也有解决办法。只要在设置属性 `spelllang` 时，给它同时带上多种语言即可（语言之间用逗号分开），这样 Vim 就可以用多种语言来进行语法检查。

```
:set spelllang=en,da,de,it
```

Vim 将会按照顺序，轮流根据每一种语言检查单词的拼写是否正确。如果单词对某一种语言来说是正确拼写的，那它就不会被标记错误。当然，这就意味着如果有一个单词的拼写本来是错误的，可是碰巧它在另一种语言中是正确的，那么它也不会被当成错误，这将会引入一个难以察觉的错误。



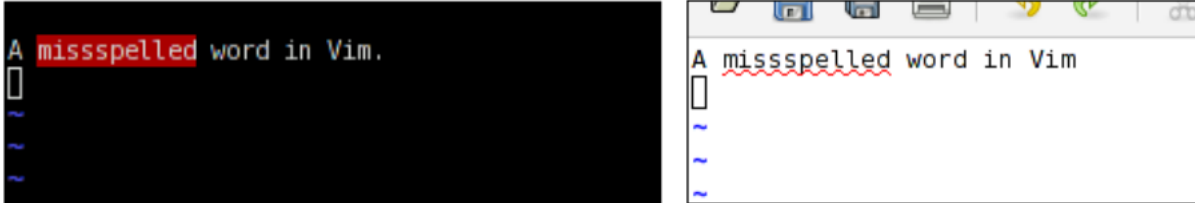
用户可以到 Vim 的 FTP 站点 <ftp://ftp.vim.org/pub/vim/runtime/spell> 上找到大量的语言包。

Vim 与 Gvim 对错误单词的标记方法稍有不同。

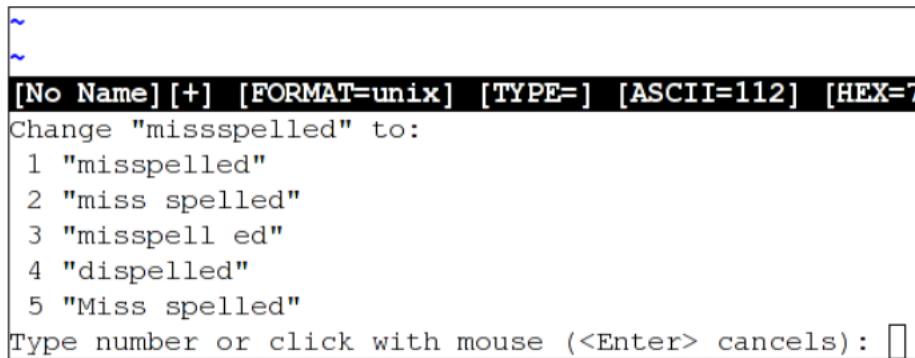
在普通的 Vim 中, 拼写错误的单词使用色彩组 SpellBad 标记 (默认情况下是红底白字)。

42

在 Gvim 中, 拼写错误的单词用红色的波浪线标记, 当然, 这也可以通过设置色彩组来改变颜色 (关于如何设置色彩组见 2.3.1 节)。



无论用户是在什么时候碰到拼写错误的单词, 都可以向 Vim 询问正确的拼写方式. 为此, 需要把光标置于单词中, 并切换到普通模式 (按 *Esc*), 然后按 *z=*.



如果可能的话, Vim 将会猜测用户试图输入的单词的正确形式, 并给出一个候选列表. 在每个候选单词的前面都有一个数字, 如果其中有你想要的单词, 就输入单词前面的数字或者单词本身, 然后按 *Enter*.

Vim 会给出一长串的候选单词, 除非用户把单词完全拼写错了, 否则的话, 单词的正确形式应该就在候选单词的前 5 个中, 如果用户不想看到一长串列表, 那么他就可以用命令

```
:set spellsuggest=width
```

来限制列表的长度, 其中 *width* 是列表的最大长度。

43

2.8.4 添加帮助性的工具提示

在 2.7 节, 我们学习了如何利用工具提示, 使得在占据较少空间的情况下, 存放更多的信息, 以此为基础, 我们将把工具提示应用在编辑器的其中地方。

编辑区是 Vim 中面积最大的部分, 所以, 为什么不利用工具提示, 在编辑区中添加一些额外的信息呢?

在 Vim 中, 编辑区的工具提示称为 *balloons*, 只在当鼠标停在组成这个单词的某个字符的上方时, 提示信息才会显示出来. 为了使用 *balloons*, 用户必须知道下面这几个命令:

- 第 1 个命令是在 Vim 中启用 *balloons*:

```
:set ballooneval
```

- 第 2 个命令告诉 Vim 在显示信息之前需要等待多长时间 (默认是 600 毫秒):

```
:set balloondelay=400
```

- 最后一个命令是设置信息的内容:

```
:set balloonexpr="textstring"
```

信息的内容既可以是静态的文本字符串,也可以是函数的返回值。

为了访问鼠标所在的位置的相关信息, Vim 提供了一些变量:

v:beval_bufnr	鼠标所在区域的缓冲区个数
v:beval_winnr	鼠标所在区域的窗口的个数
v:beval_lnum	鼠标所在的行号
v:beval_col	鼠标所在的列号
v:beval_text	触发提示信息的字符所在的单词

有了这些变量之后,来看一些具体的例子。

44

示例 1

第 1 个例子基于 Vim 的帮助系统,它展示了如何编写一个简单的函数,用来显示所有的变量。

```
function! SimpleBalloon()
    return 'Cursor is at line/column: ' . v:beval_lnum .
        \ '/' . v:beval_col .
        \ ' in file ' . bufname(v:beval_bufnr) .
        \ '. Word under cursor is: "' . v:beval_text . '"'
endfunction
set balloonexpr=SimpleBalloon()
set ballooneval
```

脚本的运行结果是:

```
362 endfunction
363 function! SimpleBalloon()
364     return 'Cursor is at line/column: 363/21 in file .vimrc. Word under cursor is: "SimpleBalloon"
365         \ '/' . v:beval_col .
366         \ ' in file ' . bufname(v:beval_bufnr) .
367         \ '. Word under cursor is: "' . v:beval_text . '"'
```

示例 2

现在来看一个更高级的例子,这个例子把 balloons 应用到编辑区的特定区域上。在这个例子中,我们将会开发一个函数,这个函数可以同时显示两种 balloons 信息:

- 拼写错误的单词: 这个 balloons 信息给出候选单词。
- 折叠的文本: 这个 balloons 信息给出被折叠的文本的预览。

现在,我们需要知道什么样的函数可以帮助判断鼠标是停在拼写错误的单词上,还是停在一个折叠行(这个折叠行代表了折叠后的多个文本行)上。

为了判断单词的拼写是否正确,需要打开拼写检查功能:

```
:set spell
```

打开后,如果鼠标所在的单词有拼写错误,被调用的内置拼写检查函数 — `spellsuggest()` — 就会返回候选单词。所以,为了判断某个单词的拼写是否有错,只需要检查 `spellsuggest()` 的返回值即可。不过有一点需要注意,即使单词没有拼错, `spellsuggest()` 也可能会返回候选单词。为了解决这个问题,在把单词交给 `spellsuggest()` 之前,需要用另一个函数过滤一下,这个函数叫作 `spellbadword()`。它的作用是把光标移动到句子中第一个拼错的单词上,然后再返回这个单词。如果输入一个拼写正确的单词,那么它应该不返回任何信息。如果没有单词被输送给 `spellsuggest()`,那它就不会返回候选单词,于是,我们现在就可以检查某个单词是否有拼写错误。

检查某个单词是否在一个折叠块中更简单。以鼠标所在的行号作为输入参数,调用函数 `foldclosed()` (回忆变量 `v:beval_lnum`),它会返回鼠标所在的折叠块的第一行的行号,如果不是折叠块,就返回 -1。换句话说,如果 `foldclosed(v:beval_lnum)` 返回除了 -1 之外的其他值,那么就在一个折叠块中。

把上面提到的所有内容都搜集起来,就得到了下面这个函数:

```
function! FoldSpellBalloon()
  let foldStart = foldclosed(v:beval_lnum )
  let foldEnd   = foldclosedend(v:beval_lnum)
  let lines = []
  " Detect if we are in a fold
  if foldStart < 0
    " Detect if we are on a misspelled word
    let lines = spellsuggest( spellbadword(v:beval_text)[ 0 ], 5, 0 )
  else
    " we are in a fold
    let numLines = foldEnd - foldStart + 1
    " if we have too many lines in fold, show only the first 14
    " and the last 14 lines
    if ( numLines > 31 )
      let lines = getline( foldStart, foldStart + 14 )
      let lines += [ '-- Snipped ' . ( numLines - 30 ) . ' lines --' ]
      let lines += getline( foldEnd - 14, foldEnd )
    else
      "less than 30 lines, lets show all of them
      let lines = getline( foldStart, foldEnd )
    endif
  endif
  " return result

  return join( lines, has( "balloon_multiline" ) ? "\n" : " "
```

45

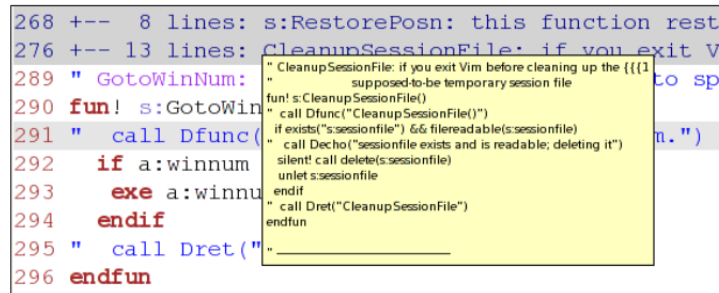
46

```
endfunction
```

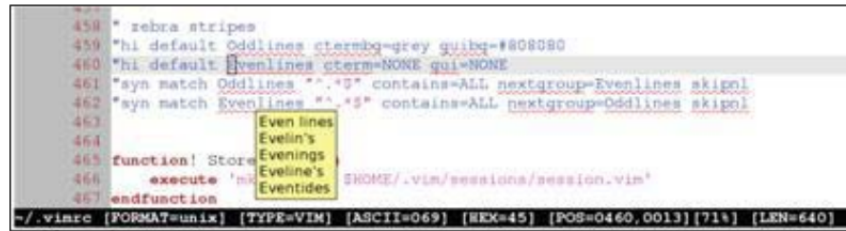
```
set balloonexpr=FoldSpellBalloon()
```

```
set ballooneval
```

这些帮助性的 **balloons** 可以极大地提高用户的工作效率, 下面的图展示了折叠块的预览效果:



如果把它应用到一个拼写错误的单词上, 得到的就是:



在第四章, 读者将会学习到如何利用行的折叠来提高 Vim 的生产力。

2.8.5 使用缩写

对于重复性工作很多人都深有体会, 但这种现象却恰恰是 Vim 想要极力避免的. Vim 的设计哲学是: 如果用户曾经为某件事物写过一次, 那就不需要再写第二次. 然而, 如果用户发现自己正在一次又一次地输入同一个内容, 那就说明他需要找到一个更好的办法.

为了避免多次输入同一个内容, 用户可以使用缩写 (abbreviations).

根据不同的使用模式, 可以使用以下三个命令来创建缩写:

- `:abbreviate`: 创建在所有模式中均可使用的缩写
- `:iabbrev`: 创建只能在插入模式中使用的缩写
- `:cabbrev`: 创建只能在命令行中使用的缩写

上面所说的三个命令都需要两个参数 — 文本的缩写形式与完整形式.

先用一个简单的例子来说明在什么情况下应该使用缩写.

用缩写来快速插入地址

在过去的几年我更换了几次住所, 所以我的一个经常性的工作就是把新住址告诉给别人. 为地址准备一个缩写并不会花上太多时间, 有了缩写之后, 就不用每次都把整个地址都写出来.

命令的具体形式是:

```
:iabbrev myAddr 32 Lincoln Road, Birminham B27 6PA, United Kingdom
```

每当我想要输入自己的地址时, 只需要键入 myAddr, 紧接着按一下空格键, Vim 就会把 myAddr 自动地扩展为完整的地址.

Vim 非常聪明, 它可以判断用户是在输入一个缩写词, 还是仅仅在输入另一个单词的某个部分, 所以它只会在输入缩写词并按了空格键之后, 才把缩写扩展为完整的字符串. 如果在输完缩写词之后, 紧跟着的是一个普通的字母, 那么 Vim 就会知道用户并不是在输入某个缩写词, 自然也就不会发生扩展. 有关缩写词 abc 的例子如下:

abc<space>	扩展
abc<enter>	扩展
123abc<space>	不会扩展, 因为缩写词现在是某个单词的一部分
abcd<space>	不会扩展, 因为在缩写词后面紧跟着的是字母
abc	不会扩展, 除非接下来输入另一个特殊字符

为了避免每次都执行一次缩写命令, 用户可以把它们放在 VIMHOME 的某个文件中, 假设文件名就叫 abbreviations.vim, 然后在文件 vimrc 使用命令 source 来使能该文件 (需确保该文件是可读的):

```
:source $VIM/abbreviations.vim
```

每当用户需要一个新的缩写词时, 可以先在编辑器中创建缩写词, 然后再把它加入到文件 abbreviations.vim.

也许读者已经意识到, 除了这些, 缩写可以应用在其他有趣的地方. 但无论如何, 这里有一些例子, 或许可以向你提供一些思路:

- 纠正常见的拼写错误:

```
:iabbr teh the
```

- 编程模版

```
:iabbr forx for(x=0;x<100;x++){<cr><cr>}
```

- 更短的命令

```
:cabbr csn colorscheme night
```

如果读者发现了一个新的缩写, 或许它看起来有点古怪, 刚开始用的时候还有点麻烦, 但是读者应该习惯这一点, 因为到最后将会发现, 它可以帮助你节省大量的时间. 用户唯一所要做的事就是把新的缩写加入到配置文件中, 然后再重新加载即可.

但是有时候缩写可能会比较恼人, 因为它自作聪明地扩展了用户并不希望扩展的单词. 比如, 用户有一个地址的缩写词是 “addr”, 但是用户其实想写的是单词 “addressed”. 在这种情况下, 缩写功能会突然冒出来, 用户也很难轻易地写出他所想要的单词.

一种解决办法是在插入单词之前调用一个函数, 该函数询问用户是否使用缩写扩展功能.

该函数的一种实现是:

```
function! s:AbbrAsk(abbr,expansion)
    let answer = confirm("use the abbreviation '" . a:abbr . "'?",
        "&Yes\n&No", 1)
    return answer == 1 ? a:expansion : a:abbr
endfunction
```

函数需要两个参数,第1个参数是缩写词,第2个参数缩写词扩展后的形式,对我们的例子来说,该函数的用法是:

```
:iabbrev <expr> addr <SID>AbbrAsk('addr', "your full address here")
```

所以,当用户写下 `addr` 时, Vim 会询问用户是否对缩写词进行扩展。

49

2.8.6 修改按键绑定

我们中的大部分人可能除了 Vim 之外,还用过其他的编辑器,正因为如此,我们经常需要用到一些特定的快捷键来完成不同的任务。

虽然 Vim 中的快捷键绑定在最开始设计时都是力求使用方便且快速,但是在某些情况下,使用用户已经熟悉的快捷键可能会更快。

为了完成这个功能, Vim 几乎可以对任何一个单独的按键进行重新绑定。

这一节将会介绍在不同的模式下,如何对 Vim 的按键绑定进行修改。

处理按键绑定的命令主要有:

- `:map` 用于普通模式,插入模式,可视模式,与命令行模式
- `:imap` 只能用于插入模式
- `:cmap` 只能用于命令行模式
- `:nmap` 只能用于普通模式
- `:vmap` 只能用于可视模式

每一个命令都需要两个参数,第一个参数是被绑定的按键,第二个参数是被绑定的命令。先看几个例子。

假设用户不太习惯在普通模式下用命令 `:w` 来保存文件,因为你已经习惯了 `Ctrl-S`,并且想继续使用下去。解决这个问题的一个映射可以是:

```
:map <C-s> :w<cr>
```

注意命令中的 `<C-s>`,这是组合键 `Ctrl+S` 在 Vim 中的表示方式。除了 `C (Ctrl)`,用户还可以用 `A (Alt)` 或 `M (Meta)`。命令末尾的 `<cr>` 是真正执行命令的地方。如果缺了它,命令只会被显示在命令行中,而不会被执行。

除了 `<cr>`,还有其他的一些用来表示按键的特殊记号,下面这张表显示了比较常见的记号:

50

按键	记号
<BS>	退格
<Tab>	制表符
<CR>	回车
<Enter>	回车
<Return>	回车
<Esc>	转义
<Space>	空格
<Up>	方向键, 上
<Down>	方向键, 下
<Left>	方向键, 左
<Right>	方向键, 右
<F1> - <F12>	功能键, F1 到 F12
#1, #2,...,#9, #0	功能键, 从 F1 到 F10
<Insert>	插入
	删除
<Home>	返回到行的开始
<End>	返回到行的结束
<PageUp>	下一页
<PageDown>	上一页

如果用户只可能在插入模式下才会保存文件, 而且在保存之后想要继续在插入模式下工作, 那就执行:

```
:imap <C-s> <esc>:w<cr>a
```

51

之后, *Ctrl+S* 就被映射成几个按键的组合. 首先, 按下 <esc> (*Escape* 键), 使 Vim 从插入模式进入到普通模式. 然后, 使用 :w<cr> 完成实际的文件保存操作. 最后, 用 a 再回到插入模式, 并将光标移到行的末尾.

用户可以通过映射来适应大多数应用程序的标准的复制/粘贴/剪切/保存快捷键, 具体来说, 可以这样做:

```
" save file (ctrl-s)
:map <C-s> :w<cr>

" copy selected text (ctrl-c)
:vmap <C-c> y

" Paste clipboard contents (ctrl-v)
:imap <C-p> <esc>P

" cut selected text (ctrl-x)
:vmap <C-x> x
```

如果是 Gvim, 甚至可以打开 Save-as 与 Open 的对话框:

```
" Open new file dialog (ctrl-n)
:map <C-n> :browse confirm e<cr>

" Open save-as dialog (ctrl-shift-s)
:map <C-S-s> :browse confirm saveas<cr>
```


用户可能会遇到这样一种情况: 组合键的第一个按键已经绑定到了 Vim 的某个功能上. 举个例子, 按键 \$, 它的功能是把光标移到一行的末尾. 用户可能希望为 \$1 新绑定一个功能, 为 \$2 绑定另一个功能, 如此等等. 具体的形式是:

```
:map $1 :MyFunction1()<cr>
:map $2 :MyFunction2()<cr>
```

现在, 如果用户按下了 \$, Vim 就会等待一秒, 等待另一个按键被按下, 如果在这一秒之内没有按键按下, Vim 就会执行原来的功能: 把光标移到一行的末尾. 相反, 如果在 1 秒之内用户按下了 1, Vim 就会执行函数 MyFunction1().

有了按键映射功能, 用户就有了一个强大的工具来根据自己的需要, 对 Vim 进行修改.



关于映射的更多的帮助信息, 可以查看 Vim 的帮助系统: :help key-mapping.

52

2.9 小结

在这一章, 我们学习了如何根据自己的具体需求来配置 Vim.

首先学习了字体与配色方案的基本修改方法, 通过它们来修改 Vim 的外观.

接下来, 讨论了如何使用颜色来标记搜索匹配, 从而使得它们更容易被识别.

为了最大限度地利用 Vim, 用户通常将大部分的区域用于编辑, 而将小部分区域用于其他功能. 因此, 我们介绍了如何配置状态行与标签列表, 使得它们占用更少的空间, 同时提供更多的信息.

虽然菜单栏与工具栏不是非常重要, 而且还会占用本来就有限的屏幕空间, 但是它们仍然可以提供很大的用处. 在这一章, 学习了如何在菜单栏中添加自定义的菜单, 以及如何在工具栏中添加图标.

为了让编辑区更符合用户的需求, 可以对它进行许多修改. 这一章简单地讨论了编辑区, 叙述了如何获得一个更好, 更具视觉效果的光标, 以及如何在编辑区中显示行号.

我们还介绍了在 Vim 中如何进行拼写检查, 从而避免拼错单词. 如果拼写检查也无法纠正错误, 还可以使用缩写.

最后, 介绍了按键绑定, 这样用户就可以继续使用其他编辑器的快捷键而养成的习惯.

现在, 用户已经拥有了一个完全定制化的 Vim, 接下来, 用户可以更加深入地学习如何在多个文件中更快速地跳转.

第三章 快速导航

53

同时应付多个文件可能是一件非常麻烦的事, 有时候, 用户可能会花更多的时间来定位文件, 而不是编辑.

Vim 的处世哲学是不浪费用户的宝贵时间, 所以它提供了许多用于定位文件的方法.

这一章将会介绍 Vim 如何帮助用户在多个文件中导航, 无论此时是在处理 1 个文件, 还是 50 个文件. 其中的某些方法使用标记, 以便于稍后返回到该区域, 还有些方法使用搜索来定位目标.

这一章包含的内容有:

- 在单个文件中更快地导航
- 在 Vim 帮助系统中更快地导航
- 在多个缓冲区中更快地导航
- 使用 Vim 文件浏览器, 以便更快地搜索文件
- 文件内搜索
- 使用 vimgrep 在多个文件或多个缓冲区中搜索
- 利用标记来导航
- 通过符号来得到更好的概览

学习后这一章之后, 用户的导航速度将会有质的提升, 在搜索文件时也不会再遇到什么问题.

54

3.1 在文件内更快地导航

有时候, 即使是一件最简单的工作 — 比如在一个单独的文件中导航 — 也有优化的空间. Vim 提供了几种在文件内导航的方法, 这些方法可以根据文件的内容和组织结构而加以调整. 其中有些方法非常简单, 而另外一些则比较复杂.

3.1.1 基于上下文的导航

在大部分情况下, 正在编辑的文件是有结构的. 如果是普通的文本文件, 那么文件的结构可以是段落, 语句, 单词, 在另外的一些场合中, 还有可能是函数, 代码块和代码行.

Vim 支持根据文件的结构, 在文件中跳转. 它还提供了一些按键绑定, 从而可以更方便地跳转到某个特定的位置上.

先来看一些例子:

- 在普通的文本文件中移动
- 在代码文件中移动

在普通的文本文件中移动

假设用户正在编辑一个普通文件文件, 此时光标正停留在一个句子的中部, 而用户突然意识到自己忘了把本段的第一个字母大写. 虽然用户可以通过方向键, 或 `h`, `j`, `k`, `l`, 把光标移到段落的首字母. 然而, 在普通模式下, 直接按下面这个按键可以得到更好的效果:

```
{
```

按完这个按键之后, 光标已经停在了段落的开头, 或者是段落正上方的空行 (如果有的话). 现在, 用户可以通过按下 `Esc` 进入到普通模式, 再按 `{`, 把光标移到段落的开始. 与此类似, 用户只要按下和 `{` 相对的按键, 即:

```
}
```

就可以把光标移到段落的末尾.

也许用户并不是在段落的末尾工作, 而是在修改段落中的某些错误. **Vim** 可以记住用户之前修改过的地方 (实际上, **Vim** 可以记住最近 999 个被修改过的地方), 因此用户可以通过查询这些信息, 从而回到正确的位置. 在普通模式下执行下面这个命令:

```
g,
```

执行该命令几次, 就可以遍历之前修改过的地方. 和 `{` 一样, 它也有一个相反的命令, 用于反向遍历之前修改过的地方, 这个命令是

```
g;
```

如果没有更多的地方可供遍历, **Vim** 就会发出一个警告.

还有一种情况是, 用户并不是在段落的开头忘记了大写字母, 而是在句子的开头, 对此, **Vim** 也提供了一对命令, 用来把光标移到句子的开始与末尾, 这对命令是:

- `(`: 移到句子的开头
- `)`: 移到句子的末尾

Vim 不希望用户在移动光标上花费太多的时间, 虽然用户可以通过方向键来遍历字母, 从而在单词间移动, 但是 **Vim** 还是认为这太浪费按键了. **Vim** 提供了一组命令, 用于在单词间移动, 比如:

- `w`: 移到下一个单词的首字母
- `b`: 移到前一个单词的首字母
- `e`: 移到单词的末尾

这些命令可以互相组合, 比如, 用户想要移到下一个单词的末尾, 只需要执行:

```
we
```

对于单词的定义, **Vim** 有两套标准:

- 一个 **word** 由字母, 数字, 破折号, 下划线组成

- 一个 **WORD** 由非空白字符 (除了制表符与空格) 组成

前面提到的命令用于 **word**, 当然, **WORD** 也会有相应的命令, 只不过使用的是大写形式 (比如用 **W** 移到下一个 **WORD** 的首字母).

56



如果读者希望在一行内多次执行本小节中提到的命令, 只需要在执行命令前加上一个数字即可, 这个数字表示命令执行的次数. 例如, `5w` 表示光标向前移动 5 个单词.

在代码文件中移动

和普通文本文件相比, 代码文件并没有段落或句子上的概念, 它包含的是大量的结构和块, 其中每一个结构或块都有特定的上下文含义. 一个简单的例子是:

```
if (a == b)
{
    print "a and b are the same"
}
```

代码中, 带有 `print` 的行在 `if` 块的上下文环境中.

因为 **Vim** 深受众多程序员的喜爱, 所以它提供了许多在代码中跳转的命令, 它们的共同点是, 目标位置和原来位置的代码之间存在着一些语境上的联系.

一个简单的例子可以是 **C** 语言中的 `#if-#else-#endif` 代码块, 这三个元素分别处于代码块的开始, 中间, 和结束.

如果用户此时正位于 `#if` 所在的行, 执行命令:

```
%
```

就可以跳转到 `#else` 所在的行, 此时再按一次 `%`, 又会跳转到 `#endif` 所在的行, 再按一次 `%` 就会回到最初的 `#if`.

Vim 无法识别所有的编程语言的构造, 不过在默认情况下, 它可以识别 **C** 语言的大部分结构. 除此之外, 它还可以识别出大部分编程语言的普通代码块 — 代码块通过圆括号与花括号定界 (例如, `{` 标出了块的开始, 而 `}` 则表示块的结束).

57



如果用户希望 **Vim** 能够识别其他更多语言的构造, 可以通过安装插件 `matchit` 来实现, **Vim 7.0** 以上的版本已经安装了该插件, 不过也可以到 <http://www.vim.org/scripts/> 上获取.

通过对程序员如何使用圆括号/花括号的简单了解, **Vim** 向用户提供了几个有用的导航命令. 也就是说, 只要代码使用了圆括号/花括号来标记一个块的开始, 并且使用相应的结束符号来标记块的结束, **Vim** 就能识别该代码块.

假设用户现在正在某个函数内, 这个函数包含多行代码, 而用户想要跳转到函数的开头. 在大部分情况下, 包围函数体的花括号是光标当前所在位置的最外层的花括号 (假设用户正在编写该函数). 于是, 对 **Vim** 来说, 为了跳转到函数的开头, 只需要找到最外层的那对花括号, 然后再跳到开括号即可.

```
function myExample() {
    ...many lines of code...
```

```

/* cursor is placed at the beginning of this line */
...many lines of code...
}

```

在上面的例子中, 命令 % 把光标移动到闭括号, 再按一次 % 就可以跳到开括号. 但是, 如果此时光标正处于另一对花括号的内部, 又该如何? 在这种情况下, 命令 % 只能让光标在这对花括号之间移动, 而无法移动到函数的开头.

Vim 还提供了其他一些方便的命令:

- [[与][: 向后/向前移动到下一节的开头 (比如函数的开头)
- [] 与]]: 向后/向前移动到下一节的结束 (比如函数的末尾)

多次执行这些命令可以让光标移动到下一节/上一节的开头/结束, 这样的话, 循环遍历文件中的函数就方便多了.

如果文件中含有两个或更多的函数, 而时光标位于第 1 个函数的开头, 按下 [两次可以把光标移动到下一个函数的开头, 以此类推. 如果想要回到前一个函数中, 只需要按下]], 光标就回到了前一个函数的开头.

58



需要注意的是, 在大部分的面向对象语言中, 类的开头与结束通常是最外层的区段.

很多时候, 用户只是想要跳转到当前块的起始处 (例如, while 循环的开始), 因为块内的局部变量都在这里定义, 对于这个需求, Vim 也有对应的一套命令:

- [{: 跳转到块的开始
-]}: 跳转到块的结束

如果是注释块, 则不会被括号所包围, 因此 Vim 也就无法利用括号来跳转到块的开始或结束.

为了处理注释块, Vim 提供了一些特殊的移动命令:

- [/: 跳转到注释块的开始
-]/: 跳转到注释块的结束

在默认的情况下, Vim 并不支持所有可能的注释格式, 它支持的注释格式主要是 C 语言 (/* */), C++ (//), 和大多数的脚本语言 (#). 然而, 如果用户想要添加对新语言语法的支持, 那么让 Vim 支持它的注释格式也是可以办到的.

有时候, 当用户在编写某小段代码时, 很有可能会忘记某个变量最初是如何定义的. Vim 提供了一个用于查看变量定义 (或变量第一次出现的地方, 如果是解释型语言 — 比如 Python — 就可能有这个需求) 的命令, 不过前提是变量是在当前文件内定义的. 当光标位于变量名上时, 按下下面这个命令, 就可以跳到变量的声明位置:

```
gd
```

这个变量非常容易记忆, 它可以看成 “Goto Declaration” 的缩写.

当执行这个命令时, Vim 所做的操作是跳到当前区段的开始 (回忆命令 [), 因为这里是定义局部变量的通常位置, 然后, Vim 在文件中向前搜索变量名第一次出现的地方. 如果在到达搜索开始的地点时还没有找到, 那就跳到文件的第 1 行, 再向前搜索变量的全局定义. 如果还是没有找到, Vim 就会在文件内执行 * 搜索 (更多的关于 * 搜索的内容可以参考 3.5 节).

59

如果用户已经知道变量是在全局定义的, 又或者是想要查找变量的全局定义, 那么可以使用 Vim 的一个命令, 该命令从文件的第 1 行开始查找, 而不是在当前区段内查找. 这个命令是:

gD

Vim 非常聪明, 它会自动忽略注释块中的变量引用, 因为这里绝不可能出现变量的声明。

如果 Vim 找到了变量的定义 (或者是该变量在文件中第一次被用到的地方), 那么光标就会跳转到该位置。



在执行 gD 之前输入一个 l (即 lgD), 就可以让 Vim 忽略被 {} 包围的代码块的匹配, 该代码块出现在当前的光标位置之前 (例如, 在文件早先位置定义的另一个函数)。

3.1.2 在长行内导航

有些人喜欢回绕显示过长的行, 而有些人则希望单行显示, 即使过长的行会跑到边界之外。从笔者个人来说, 我更倾向于回绕显示, 因为这样可以更方便地看到文本的全貌。不过这有时会让人感到很讨厌。如果用户面对着一个回绕的长行, 那么回绕的部分在视觉效果上就像新的一行那样显示。这本来并没有什么问题, 但是如果在这种回绕的长行内移动光标 — 比如用 j/k — 那么 Vim 就会忽略回绕的部分, 而直接把光标移动到真正的下/上一行。如果用户不太喜欢这种行为, 也可以通过一个小技巧解决。

假如用户希望在按住 Alt 的同时, 再用方向键上/下来移动光标, 那么 Vim 就应该按照视觉上的行 — 而不是实际的行 — 来响应。为了完成这样的效果, 需要在文件 vimrc 中添加几行按键映射:

```
map <A-DOWN> gj
map <A-UP> gk
imap <A-UP> <ESC>gki
imap <A-DOWN> <ESC>gji
```

映射只能在普通模式与插入模式下使用。如果用户希望在没有按住 Alt 的情况下也能正常工作, 只需要删除掉按键组合中的 A- 部分即可 (例如把第一条命令改成 map <DOWN> gj)。

60

3.2 在 Vim 帮助中快速地导航

Vim 自带了非常完善的帮助系统, 用户此时应该已经用到了这个功能。然而你可能不知道的是, Vim 的帮助系统支持超链接, 就像网页中的那样。有两种超链接 — 主题链接标记成 “some subject”, 而选项链接标记成 “option”。

一个主题链接引用到帮助系统中一节的开始, 而选项链接则会把你带到一个特定选项的描述。如果把光标移动到一个链接上, 再按下 **Ctrl+J**, 就可以跳转到链接的目标地址, 无论该链接是什么类型。这个功能非常方便, 但是如果用户使用的不是英语键盘布局, 那么可能就无法用单个按键来表示 J, 在这种情况下, 重新映射按键会比较好。在一个网络浏览器中, 用户可以跳转到一个链接上, 再按下 **Enter**, 为了完成这样的效果, 用户可以执行:

```
nmap <buffer> <CR> <C-J>
```

如果用户正在使用网页浏览器, 并且想要回到之前浏览过的页面中, 这个操作可以通过按退格键来完成。如果在 Vim 帮助系统上也加上这个功能, 那就再好不过了。下面这个映射可以用来完成这个功能:

```
nmap <buffer> <BS> <C-T>
```

通过易于记忆的按键绑定, 在 Vim 帮助系统中前进或后退。接下来, 不妨添加几个导航键, 用于寻找当前打开着的帮助文件中的下一个/前一个主题或选项链接。有了这些导航键的帮助, 用户就可以在帮助文件中快速地滚动搜索, 直到找到想要的信息。

```
nmap <buffer> o /'[a-z]\{2,\}''<CR>
nmap <buffer> O ?'[a-z]\{2,\}''<CR>
nmap <buffer> s /\|\\S\+\\<CR>
nmap <buffer> S ?\\|\\S\+\\<CR>
```

现在, 用户可以按下 `o` 前进到下一个选项链接, 按 `s` 前进到下一个主题链接, 后退操作是类似的, 只不过把小写字母改成大写字母 — 按下 `O` 后退到前一个选项链接, 按下 `S` 后退到前一个主题链接。



为了防止按键映射之间互相干扰, 用户可以把它们添加到文件 `help.vim` 中, 并把该文件存放到 `$VIMHOME/ftplugin/`。

到目前为止, 对于 Vim 帮助系统导航的提升只剩下最后一点工作: 需要一种快速打开帮助系统的方法。通常来说, 当用户按下 `F1` 时, 帮助系统就在默认页面打开。然而, 如果在按下 `F1` 时, Vim 能够自动搜索光标当前位置下的单词, 那就方便多了。完成这个功能的按键映射是:

```
:map <F1> <ESC>:exec "help ".expand("<cWORD>")<CR>
```

这个例子稍微有点难以理解。命令 `:help` 用于查找待搜索的命令, 而跟在它后面的命令并不会被解释执行, 正因为如此, 所以要用 `:exec` 把命令包裹起来。

为了获取光标下的单词, 用到了 `cWORD`。大写部分 `WORD` 表示: 除了空白字符 (空格与制表符) 外的所有字符都可以是单词的构成成分。我们需要这样的单词定义, 因为 Vim 的命令除了字母数字, 还可以包含其他特殊字符 (比如说, 查找 `<cWORD>` 的帮助信息)。

这个按键映射可以从帮助系统的外部调用, 也可以把它写到 `vimrc`, 但不能写到 `help.vim`。

3.3 在多个缓冲区中更快地导航

许多情况下, 用户并不是在处理一个文件, 对于每一个打开过的文件, Vim 都会打开一个缓冲区。缓冲区可以被显示或隐藏, 这意味着为了找到某个文件, 用户必须找到与它对应的缓冲区。

当然, 用户可以打开缓冲区列表, 然后在列表中一个个搜索。为了打开缓冲区列表, 需要执行:

```
:buffers
```

这份列表是不可交互的。为了选择期望中的缓冲区, 用户需要查看缓冲区列表左边的数字, 这个数字表示文件所在的缓冲区号码。有了这个号码, 用户就可以直接打开与该号码对应的缓冲区, 具体的命令是:

```
:buffer N
```

`N` 表示缓冲区的号码。

这种打开缓冲区的方法并不总是最高效的。用户还可以用下面这两个命令来循环遍历所有的缓冲区:

- `:bnext`
- `:bprevious`

虽然这些命令拥有缩写形式 (`:bn` 与 `:bp`), 但它们仍然需要在普通模式下输入并执行。这意味着为了执行这两个命令, 至少需要按 5 个按键, 还是不太方便。

为了更快地遍历缓冲区, 用户可以把下面这两个按键映射加入到 `vimrc`:

```
map <C-right> <ESC>:bn<CR>
map <C-left> <ESC>:bp<CR>
```

上面两行命令的功能是用组合键 **Ctrl+Left** 打开前一个缓冲区, 用 **Ctrl+Right** 打开下一个缓冲区. 于是, 在按住 **Ctrl** 的同时, 重复地按下左/右方向键, 就可以快速地遍历缓冲区列表.



如果用户只想在当前文件和前一个文件之间切换, 可以用 **Ctrl+6** (插入模式下用 **Ctrl+o Ctrl+6**) 或者 **:e#**.

3.4 快速打开引用过的文件

在许多程序设计语言中, 你可以在当前文件中包含其他文件, 这样就可以把一个文件的内容切分后放到多个文件中. 文件的包含类似于:

```
#include "somefile.h"
```

在上面的示例中, `somefile.h` 是被包含的文件的名字.

如果能有一个简便的方法用来打开被包含的文件, 那就太棒了. **Vim** 的确提供了这样的命令. 把光标移动到你要打开的文件名上, 再在普通模式下执行:

```
gf
```

可以把这个命令记成 “goto file”. 执行这个命令时, **Vim** 会在下列几个地方寻找文件:

- **Vim** 首先在选项 `path` 中定义的, 并且相对于当前打开着的文件的目录中寻找.
- 如果没有找到, **Vim** 就使用函数 `suffixadd`, 查看是否可以通过加上一个后缀来搜索到文件 (比如, 在文件名后面加上 `.c`)
- 如果还是没有找到, **Vim** 就使用表达式 `includeexpr`, 把文件名转换成更像文件名的形式 (例如, 把 `java.com.http` 转换成 `java/com/http.java`)

如果 **Vim** 找到了文件, 它就会在当前缓冲区中打开它, 如果没有找到, 就返回一条错误消息. 如果当前所在的缓冲区有更新的内容还没有被保存, 或者是其他正在进行的工作阻止了 **Vim** 丢弃当前打开着的文件, 那么 **Vim** 就无法打开另一个文件. 有时候这相当恼人, 幸好有办法可以阻止这种情况发生. 只要把下面这行命令添加到 `vimrc`, **Vim** 就会在另一个缓冲区中打开新文件, 这样 **Vim** 就不需要丢弃当前打开着的文件:

```
:map gf :edit <cfile><CR>
```

这行代码覆盖了命令 `gf` 原来的功能, 取而代之的是用命令 `:edit` 打开光标下的文件. 如果文件不存在, 命令就会打开一个新的空缓冲区.



使用 `gf` 时, 如果想让 **Vim** 支持带空格的文件名, 可以把 `set isfname+=32` 添加到 `vimrc`, 32 是空格字符在 **ASCII** 表中的十进制值.

3.5 搜索即可得到

很多人都有过这种感觉: 依稀记得自己在某个地方给写错了, 可就是想不起来是在哪里. 解决这种问题的办法通常是搜索.

Vim 当然也可以搜索. Vim 的搜索可以分成三类:

- 在当前文件中搜索
- 在多个文件中搜索
- 在帮助系统中搜索

下面的一节将会向读者展示这三类搜索的使用秘诀.

64

3.5.1 在当前文件内搜索

也许用户正在编辑的文件并不长, 但是要想通过肉眼在字里行间搜索仍然是一件非常痛苦的事情. Vim 提供了几种在文件内搜索的方法, 先从一些简单的例子开始.

示例 1: 搜索单词的下一出现

用户知道在单词 “someWord” 的附近有自己感兴趣的内容, 为了找到它, 可以在普通模式下执行:

```
?someWord
```

该命令在文件内向后搜索问号右边的单词的第一次出现. 如果光标当前是在文件的末尾, 用它来搜索单词正好合适. 然而, 如果当前光标是在文件的开始, 那么向前搜索可能会更合理一点. 为了向前搜索单词, 只需要把问号改成斜杆:

```
/someWord
```

被搜索的单词可能会多次出现, 第一次找到的地方可能不是用户想要的. 不用担心, 按下 `n` 就可以按照搜索方向, 前进到单词的下一出现. 如果想要临时改变搜索方向, 按下 `N`, 就会跳转到单词的前一次出现.

如果用户想要再搜索一次相同的内容, 没必要再次输入完整的单词, 只需要执行 `??` 或 `//`.

如果用户设置了 `incsearch`, 在用户输入待搜索单词的同时, Vim 就开始执行搜索任务, 并把光标移动到搜索到的单词所在的位置. 每当用户输入待搜索单词的下一个字符时, 光标就会跳转到与当前搜索文本相匹配的下一出现/前一次出现.



在输入待搜索的文本后, 用户必须按下 `Enter`, Vim 才会真正地去执行搜索任务, 否则的话, 光标会回到最初的位置. 为了取消搜索, 并回到最开始的地方, 只需要按下 `Esc`.

65

搜索光标下的单词

如果用户已经很接近待搜索单词的某次出现, 但它并非你所想要的那次出现. 或者是你想要遍历某个单词所有出现过的地方, 并且这个单词已经写出来了, 那就没有必要把单词再打一遍, Vim 提供了应对这种情况的命令. 把光标移动到待搜索的单词上, 再在普通模式下按下下面任意一个键:

```
#
*
```

第一个命令搜索光标下的单词的前一次出现, 第二个命令搜索单词的下一次出现. 多次执行命令可以重复地跳转到单词的下一次/前一次出现. 有了这两个命令的帮助, 遍历某个单词的所有出现就方便多了.

也许用户想要搜索的内容并不是一个完整的单词, 只是单词的一部分. 对此 Vim 也有解决办法, 在普通模式下执行下面命令中任意一个:

```
g#
g*
```

现在, 光标不仅会跳转到单词的下一次出现, 还包括包含该单词的单词. 例如: 把光标移到单词 “foo” 上, 再按下 g#, 如果上文含有单词 “foobar” 与 “food”, 光标就会陆续跳转到这两个单词上.

3.5.2 在多个文件内搜索

也许用户想要搜索的内容并不在当前文件内, 甚至不知道在哪个文件中. 在类 Unix 系统中, 比如 GNU/Linux, 用户可以用 Shell 命令 grep, 在多个指定的文件内搜索特定的单词或模式. Windows 操作也有类似的命令: FIND 和 FINDSTR, 但是用户很少使用它们. 为了向所有的 Vim 用户 (无论他们使用的是什么操作系统) 提供在多个文件内搜索的方法, Vim 配备了自己的 grep 命令. 命令的使用方式是:

```
:vimgrep /pattern/[j][g] file file2... fileN
```

这个命令需要两个参数. 第一个参数是用户想要搜索的模式, 用户既可以使用 Vim 的正则表达式, 也可以直接输入单词. 模式需要用一对 / 括起来, 在右边的 / 后面, 用户可以添加一个标志: j 或 g. 标志可以帮助用户选取结果, 以及如何展示搜索结果.



除了 /, 你也可以用任意一个非 ID 字符. 一个非 ID 字符指的是没有定义在选项 `isindenta` 中的字符.

^aVim 中找不到该选项 — 译者注

如果添加了标志 g, 那么搜索结果就会包含一行内的所有匹配. 意思是说如果在同一行内, 待搜索的模式出现了三次, 那么在搜索结果中该行就会显示三次. 如果添加标志 j, 那么用户将看不到搜索的结果, 其实 Vim 是把搜索结果更新到了 `quickfix` 列表中, 以便于后面的检索 (关于 `quickfix` 列表的更多信息, 参考 `:help quickfix`). 如果没有标志 j, 那么光标将会直接跳转到第一个匹配, 剩下的匹配结果则添加到 `quickfix` 列表.



为了显示 `vimgrep` 搜索结果的 `quickfix` 列表, 可以使用 `:clist` 命令, 或者用 `:cnext` / `:cprevious` 跳转到前一个/后一个匹配.

`vimgrep` 的第二个参数是待搜索的文件列表. 文件列表可以由单独的文件名, 或文件名列表, 或包含通配符的模式 (比如 `*.c *.h`) 组成. 用户还可以使用双星号通配符 `**`, 比如用 `**/*.c`, 来搜索当前目录中的所有 C 文件, 并递归搜索子目录.

```

:clist..
1 Desktop/diverse/fpix-0.90.1/driver/finepix-main.c:309 col 95: dev_err
   rame. Please, report to driver maintainer.\n");
2 Desktop/diverse/fpix-0.90.1/userspace/fpix-stress-v4l2.c:3 col 13: *
3 Desktop/diverse/fpix-0.90.1/userspace/fpix-stress-v4l2.c:19 col 5: in
4 Desktop/diverse/fpix-0.90.1/userspace/fpix.c:3 col 13: * Public domai
5 Desktop/diverse/fpix-0.90.1/userspace/fpix.c:44 col 5: int main(void)
6 Desktop/diverse/fpix-0.90.1/userspace/fpixtest.c:3 col 13: * Public d
7 Desktop/diverse/fpix-0.90.1/userspace/fpixtest.c:16 col 5: int main(v
8 Desktop/diverse/fuji-finepixa310-test/stream.c:255 col 5: int main(in
9 bin/bin2iso.c:29 col 5: int main( int argc, char **argv )
10 bluemote/bluemote.c:56 col 5: int main(int argc, char *argv[])
11 bluemote/bluemote.c:147 col 11: if(init_mainmenu()==-1) continue;
12 bluemote/bluemote.c:261 col 10: int init_mainmenu()
13 bluemote/bluemote.c:291 col 22: while((ret = init_mainmenu())==0);
14 bluemote/mouse.c:75 col 1: main (int argc, char **argv)
15 btsc/a2play.c:536 col 5: int main(int argc, char *argv[])
:vimgrep /main/ **/*.c

```

67

3.5.3 搜索帮助系统

有时候需要从 Vim 中得到某些帮助信息, 此时用户可能并不确切地知道应该查找哪些信息. 当然, 你可以在整个帮助系统中从头开始查找, 不过, Vim 的帮助系统包含众多的文件, 以及数以千计的关键词.

为了解决这个问题, Vim 提供了一个专门的命令. 在前一节中, 用的是 vimgrep, 而对于 Vim 帮助系统, 则用的是下面这个命令:

```
:helpgrep pattern [@LANG]
```

命令需要一个必填参数 — 待搜索的模式 — 还有一个是可选参数, 用于限定语言, 先来看一个例子. 比如说用户需要一些关于自动补全的帮助信息, 但是不知道从哪里开始查找. 因为用户只了解英文, 所以希望你希望帮助信息也是英文的. 完成这个搜索的命令是:

```
:helpgrep completion@en
```

命令所执行的操作是在所有的英文 (en) 文档中搜索单词 completion. 搜索完成后, 光标会跳转到第一个匹配, 剩下的匹配结果则添加到 quickfix 列表, 以便稍后检索.



如果用户想使用 location 列表, 而不是 quickfix 列表, 就用 :lhelpgrep 替换掉 :helpgrep.

实际上, 命令 helpgrep 并没有遍历所有的文档, 而是使用了 tag list 来搜索模式. Tag list 并不会自动生成, 所以, 如果用户安装的插件带有自己的文档, 那就必须使用下面的命令:

```
:helptags /path/to/documentation
```

把 /path/to/documentation 替换成新文档的安装位置, 但是为了能让 Vim 找到文档, 新文档必须放在 Vim 选项 runtimepath 定义的几个目录的 docs/ 子目录中 (参考 :help 'runtimepath').

68

3.6 标记位置

有时,当用户在编辑文件的某一行时,需要跳到另一个地方查看一下.之后,可能很难找到原来的那一行.如果能在离开前作一下标记,这样的话,找到离开时的位置就方便多了.为此,Vim 提供了一些工具用于完成这个功能.这些工具可以分成两类:

- 可见的标记
- 隐藏的标记

3.6.1 可见的标记 — 使用符号

在 Vim 中,可以用一个可见的标记 — 符号 — 来标记一行.一个符号指的是一个标记,它会显示在编辑器最靠左的列中.



如果用户想改变显示符号的列的颜色,可以使用下面这个命令:

```
:highlight SignColumn guibg=darkgrey
```

根据所使用的 Vim 版本 (控制台或 GUI) 的不同,符号或者是一些字符的组合 (比如 >>), 或者是一个图标.为了使用符号,需要作一些设置.如果把设置信息写在 vimrc 中,就不用每次都设置一遍.

第一件要做的事是定义你想用的符号.定义符号的命令是:

```
:sign define name arguments
```

arguments 可以是下面几种之一:

- linehl: 用于标记行的色彩组
- text: 该文本将作为符号,显示在控制台 Vim 中 (比如 >>!! 或 ++). 每一个符号最多可以用两个字符.
- texthl: 用于标记符号文本的色彩组
- icon: 图标的完整路径,该图标可用在 Gvim 的符号中.图标应该足够小,小到能够放到两个字符的空间中.图标文件的格式可以是位图文件,不过最好是 .xpm.

一个简单的例子是:

```
:sign define information text=!> linehl=Warning texthl=Error icon=/path/to/information.xpm
```

把符号定义好,并且把定义的命令添加到 vimrc 后,我们已经准备好把该符号放到文件中的某个位置,放置符号的命令是:

```
:exe ":sign place 123 line=" . line(.) . "name=information file=" .  
expand("%:p")
```

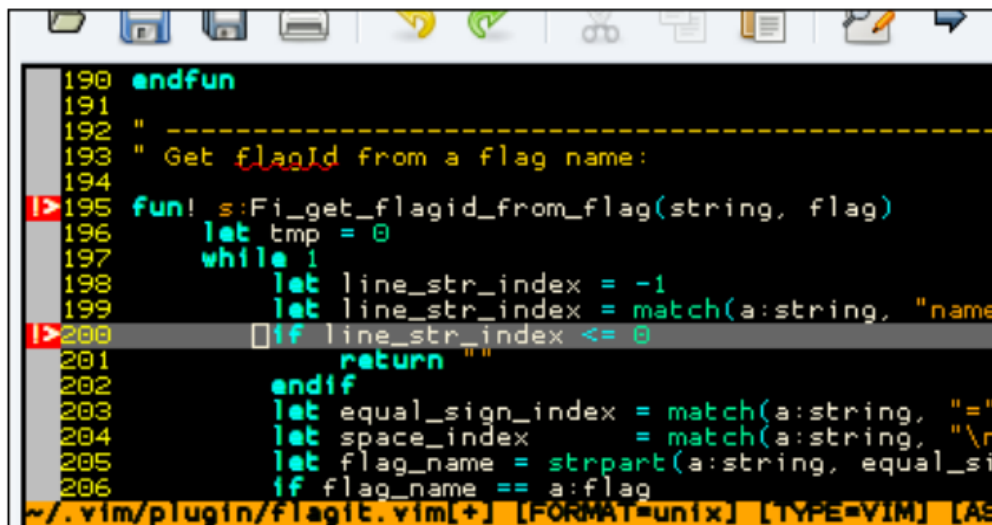
用户可以把数字 123 替换成任意一个数字,它将作为这个符号的 ID.

正如读者所看到的那样,放置符号的命令写起来有点麻烦,但可以把它映射到一个键上.

命令的效果是把名字为 information,且 ID 号为 123 的符号放置到当前打开文件 (expand("%:p")) 的当前行 (line(.)). 映射的命令是:

```
:map <F7> :exe ":sign place 123 line=" . line(".") . "name=infomation
file=" . expand("%:p") <CR>
```

上面的命令把符号 `infomation` 映射到键 `F7`, 当你按下 `F7` 时, 符号就会被放置到当前行。



有时候, 用户想要移除符号, 用术语来讲, 叫作 “unplace” 一个符号:

```
:sign unplace ID
```

命令中的 *ID* 是放置符号时的 *ID* (在前面的例子中就是 123). 这个命令会把所有的, 用该 *ID* 放置的符号移除. 用户可能只想把当前文件中的指定符号移除, 这时候可以在命令的末尾加上另一个参数, 用来指定文件:

```
:sign unplace ID file=name
```

或者用下面这个命令从当前缓冲区中移除符号:

```
:sign unplace ID buffer=bufferno
```

命令中的 *bufferno* 是当前缓冲区的编号 (可以通过命令 `:buffers` 查看).

如果只想移除当前行的符号, 用:

```
:sign unplace
```

为了和放置符号的按键映射相对应, 把它映射到 `Ctrl+F7` 上:

```
:map <C-F7> :sign unplace<CR>
```



如果你在同一个文件中, 用同一个 *ID* 添加了多个符号, 那么上面的命令只会移除文件内最靠前的符号, 而非当前行的符号。

既然本章讨论的是导航, 那么我们也要介绍一下如何跳转到一个符号. 这种跳转在 Vim 中称为符号跳转 (sign-jumping), 所使用的命令是:

```
:sign jump ID file=file
```

命令中的 *ID* 是你想要跳转到的符号 ID, *file* 指定在哪个文件中搜索该符号. 除了 `file=`*file*, 还可以用 `buffer=`*buffer*.

同样, 如果在同一个文件/缓冲区内用同一个 ID 添加了多个符号, 那么命令只能跳转到该 ID 在文件内的第一个符号.



Paul Rouget 开发了一个 Vim 脚本, 通过它用户可以非常方便地使用符号. 脚本的下载地址是 http://www.vim.org/scripts/script.php?script_id=1580.

3.6.2 隐藏的标记

为了给当前行作标记, 用户可以使用标记 (mark). 它的使用方法非常简单, 总的来说, 它由两个在普通模式下使用的命令组成, 一个用于设置标记, 另一个用于跳转到标记. 除非打开标记列表, 否则用户无法判断某一行是否含有标记.

首先来看一下如何标记当前行. 为了标记当前行, 只需要在普通模式下按下 `m`, 后面再紧跟着某个字符, 这个字符可以是 0-9, a-z, 或 A-Z 中的一个. 例如, 如果用户执行了 `ma`, 那就是说用名字为 `a` 的标记来标记当前行. 然后, 如果用户想要跳转到标记 `a` 所在的行, 只需要按下 `'a` (单引号 + 标记名), 光标就会马上跳转到该行的开始 (如果该行是缩进过的, 那就跳转到该行的第一个非空白字符).

有时候, 光是跳转到一行的开始可能还不够, 如果能直接跳转到添加标记时光标所在的具体位置可能会更好一点. 为了达到这个目的, 需要把跳转命令中的单引号改成反单引号 (```), 于是, 跳转到标记 `a` 的命令就变成了 ``a`. 不同的标记名含有不同的意义, 它们的使用范围如下所示:

标记	用法
0-9	这个标记集来源于文件 <code>.viminfo</code> , 通常保留给 Vim 内部使用 (比如, 标记 0 表示文件最后一次退出时, 光标所在的位置. 然而, 用户可以利用这些标记来实现 “打开最近使用过的文件”.
a-z	这些标记只在当前文件内有效, 当文件关闭时删除这些标记. 如果用户此时是在当前文件的缓冲区内, 那么只能通过这些小写字母标记在文件内跳转.
A-Z	这些标记可以跨越文件使用. 即使没有打开目的标记所在的文件, 也可以直接跳转到标记所在的位置. 如果 <code>.viminfo</code> 文件是可用的, 那么这些标记就会被保存下来, 直到下一次对文件进行编辑.

用户可以通过下面这个命令来获取完整的, 正在使用的标记列表:

```
:marks
```

命令显示了每个标记所在的文件与行号. 为了删除一个或多个标记, 执行:

```
:delmarks markid markid...markid
```

具体的使用例子有:

```
:delmarks a b c
:delmarks a-c
:delmarks a f-i 1-4
```

如果用户想要删除当前缓冲区内的所有标记, 执行:

```
:delmarks!
```

当使用 **Vim** 时, 它会自动设置其他类型的标记. 其他类型的标记包括但不限于: 最后一次退出插入模式时光标所在的位置; 可视模式下被选中的文本的开始与结束; 最后一次被修改的地方.

关于如何使用标记, 以及其他类型标记的更多信息, 参考 `:help mark-motions`.

3.7 小结

这一章介绍了在文件与缓冲区内快速导航的方法.

首先介绍了如何根据文件的上下文结构, 在文件内部快速导航. 除此之外, 还讨论了如何在冗长并且折叠过的行中移动光标.

然后介绍了如何在 **Vim** 的帮助系统中导航, 还学习到了如何通过简单的按键绑定, 使得导航更加直观, 便于记忆.

既然已经知道了如何在文件内导航, 还要知道如何在多个文件与多个缓冲区之间导航. 再接下来的小节我们学习了如何快速地在缓冲区之间导航, 以及如何通过两次击键, 来打开被另一个文件所引用的文件.

导航的方式有许多种, 在接下来的小节我们学习到利用搜索, 不仅可以在打开的文件内导航, 甚至包括磁盘上的文件. 还学习了如何在 **Vim** 的帮助系统搜索某个主题的相关帮助信息.

最后, 我们学习了如何通过符号与标记在文件间跳转, 以及使用 **Vim** 时, 它会如何自动地帮助用户添加标记.

我们已经学习到了如何在 **Vim** 中迅速地导航, 可以继续阅读下一章节. 下一章将会用到更多的 **Vim** 内置功能来提高工作效率.

第四章 助推器

73

在这一章将会看到,即使是一些小小的改动,也可以极大地促进 Vim 的工作效率. 有些技巧是由 Vim 的特性提供的, 另外一些则需要用户自己编写一些脚本.

无论你把 Vim 当成一个修改配置文件的小工具, 还是把它用作某个大型开发项目的主要编辑器, 你都可以发现本章介绍的方法可以极大地提高 Vim 的使用效率.

这一章讨论的主题包括:

- 使用模版文件的模版
- 使用缩写的模版
- 使用已知单词与 tag list 的自动补全
- 使用 omnicompletion 的自动补全
- 宏与宏录制
- 使用会话
- 使用会话的项目管理
- 寄存器与撤消分支
- 折叠
- 使用 vimdiff 分析差异
- 使用 netrw 随时随地地打开文件

阅读完这一章之后, 用户使用 Vim 的工作效率应该可以提高好几个百分点.

74

4.1 使用模版

无论编辑的是哪一种类型的文件, 当打开一个新文件时, 总有一些基础性的工作需要完成. 手动完成这些基础工作是一件非常乏味的事件, 更讨厌的是每次打开一个新文件时, 都要重新再做一遍. 所以说干嘛要花这么多的时间, 来做一件使用模版就可以完成的事情?

在接下来的两节, 将会看到一些不同类型的模版. 其中一些模版特定于文件类型, 另外一些则会使用用户的输入来触发小内容模版 (比如, 程序员经常用到的代码片断).

4.1.1 使用模版文件

每次打开一个新文件时, 用户做的第一件事经常是输入某些头部信息, 当然, 所要输入的信息取决于文件的类型. 比较常见的例子包括:

- 在新的 **HTML** 文件中添加基本结构 (<html>, <head>, <body>).
- 在所有的 **C** 文件添加头部信息, 在文件 `main.c` 中添加 `main()` 函数.
- 在 **Java** 文件中添加主类.

除了这些, 读者应该还能想到其他一些例子.

那么, 怎么才能创建一个模版文件? 不妨用 **HTML** 文件作为例子来进行讲解. 这种文件的结构是静态的, 因此非常适合用模版来处理. **HTML** 模版的内容是:

```
<html>
  <head>
    <title></title>
    <meta name="generator" content="Vim" />
    <meta name="author" content="Kim Schulz" />
  </head>
  <body>
    <p>Content goes here...</p>
  </body>
</html>
```

在 `VIMHOME` 目录下创建一个新目录 `templates/`, 并把上面的模版文件保存到这个目录中, 假设把模版文件命令为 `html.tpl`. 现在, 第一个模版已经准备就绪, 不过, 创建一个新的 **HTML** 文件时, 需要加载模版文件, 为了完成加载, 把下面这行命令添加到 `vimrc` 中:

```
:autocmd BufNewFile *.html 0r $VIMHOME/templates/html.tpl
```

这个命令可以确保在新建一个 `*.html` 文件时, 模版文件的内容会自动加载到新文件中. 于是, 在开始编辑新文件前, 文件中就已经包含了模版的内容.

本来这样做没什么问题, 但是当添加的模版文件越来越多时, 用户可能会越来越讨厌每次都要往 `vimrc` 中添加一行加载命令. 所以, 把加载模版的命令写得更灵活一点:

```
:autocmd BufNewFile * silent! 0r $VIMHOME/templates/:%:e.tpl
```

命令的功能是: 无论何时打开一个新文件, **Vim** 就在模版目录中搜索以文件扩展名命名的模版. 比如, 创建一个 `index.html` 文件时, **Vim** 就在 `$VIMHOME/templates/` 中搜索名为 `html.tpl` 的模版文件.

如果没有找到指定的模版, **Vim** 就创建一个空文件.

再把模版写得更完善一点: 添加对占位符 (占位符指的是将要添加文本的地方) 的支持. 一个占位符看起来可以非常得与众不同, 这取决于用户, 我通常把它们显示成类似于 `<+KERWORD+>` 的形式. 如果在前面的 **HTML** 模版中添加上占位符, 模版的内容就变成了:

```
<html>
  <head>
```

```

<title><+TITLE+></title>
    <meta name="generator" content="<+GENERATOR+>" />
    <meta name="author" content="<+AUTHOR+>" />
</head>
<body>
    <p><+CONTENT+></p>
</body>
</html>

```

现在, 已经准备好了占位符, 接下来的工作就是在占位符之间跳转. 为了让跳转更加方便, 把跳转命令添加到 vimrc 中. 用户可能想要把组合键 **Ctrl+j** 映射到跳转命令, 因为这样做的话便于在插入模式下使用, 组合键中 j (意指 jump) 也更容易记忆. 映射组合键的命令是:

```

nnoremap <c-j> /<+.\{-1, \}+><cr>c/+>/e<cr>
inoremap <c-j> <ESC>/<+.\{-1, \}+><cr>c/+>/e<cr>

```

现在, 只需要按下组合键 **Ctrl+j**, 你就可以方便地跳转到文件中的下一个占位符, 输入文本, 然后再跳转到下一个占位符.

通过在占位符中添加关键字, 就可以提醒用户应该在占位符中输入什么样的内容.



可以通过命令 `match` 高亮显示占位符, 具体的设置命令是: `match Todo /<+.\{++>/`, 把这行命令添加到 vimrc 中 (可以把命令中的 `Todo` 替换成任何一种你喜欢的色彩组).

76

4.1.2 把缩写作为模版

上一小节介绍了如何为不同类型的文件制作模版, 这一节介绍如何为文件内容制作模版.

第二章简单地介绍了如何通过缩写来减少击键的次数, 现在把缩写应用到模版中. 回忆一下下面这个命令的作用:

```
:iabbrev match replace-string
```

我们只想让上面的命令工作在插入模式, 这是因为模式模版本来就应该工作在插入模式下. 对于 C 文件来说, 一个常见的例子是:

```
:iabbrev <buffer> for( for (x=0;x<var;x++){<cr><cr>}
```

无论在什么时候输入文本 `for(`, Vim 都会自动插入一段 `for` 循环. (可以防止手工输入的 `for` 循环被自动转换. 插入的内容看起来就像:

```

for (x=0;x<var;x++){

}

```



缩写词左边的 `<buffer>` 把命令的作用范围限制在当前缓冲区内.

正如你所看到的那样,生成的代码模版是静态的. 为了让模版更灵活一点,把前面一节介绍的占位符应用进来. 在这种情景下使用的占位符更像是一个跳转点,因此把它们简化成 <+++>. 除此之外,在插入模式模版之后,需要在光标的下一个位置上放置一个占位符,对于前面的例子来说,光标的下一个位置应该是左括号的右边.

77

为了实现这个目标,引入 !cursor! 占位符,命令的内容是:

```
iabbrev for( for(!cursor!;<+++>;<+++>){<cr><+++><cr>}<Esc>
:call search('!cursor!','b')<cr>cf!:
```

(上面的命令都在同一行上)

现在,无论何时输入 for(, Vim 都会自动插入一段 for 循环,然后把光标移动到占位符 !cursor! (当光标移过来时,占位符上原来的内容会自动被移除). 接下来,用户就可以方便地填写 for 循环的参数,并用 Ctrl+j 跳转到下一个参数的位置.

也许读者已经知道,许多编程语言都有相同的主要结构(比如 for 循环),但是它们之间的差异导致了无法使用同一个模式模版. 再回过头来看一下手上已有的东西,看看是否可以让它们识别文件类型.

前面的一节介绍了如何根据文件的扩展名来加载相应的模版文件,具体的命令是:

```
:autocmd BufNewFile * silent! 0r $VIMHOME/templates/%:e.tpl
```

对这个命令进行修改,使得 Vim 可以根据文件的类型,为模式模版自动加载适当的缩写.

为了能让命令更聪明一点,把命令的功能用函数来实现. 函数的代码是:

```
function! LoadTemplate(extension)
    silent! :execute '0r $VIMHOME/templates/'. a:extension. '.tpl'
    silent! execute 'source $VIMHOME/templates/'.a:extension.'.patterns.tpl'
endfunction
```

为了调用函数,把 autocmd 修改成:

```
:autocmd BufNewFile * silent! call LoadTemplate('%:e')
```

函数 LoadTemplate 在 \$VIMHOME 的 templates 子目录下搜索两个文件: EXTENSION.tpl 与 EXTENSION.patterns. 其中 EXTENSION 表示当前打开着的文件的扩展名. 第 1 个文件包含了特定文件类型的模版,而第 2 个文件则包含了特定文件类型的缩写命令. 如果没有找到对应的文件,那么命令 silent! 就会抑制错误信息的输出,什么也不会显示出来.

78

现在,模版文件中该包含什么样的内容完全由用户来决定.



已经有人为 Vim 编写了大量的模版系统脚本,其中大部分都是以本章介绍的概念为基础,但是增加了许多额外的功能. 除了这里介绍的之外,如何用户还想要更多的模版选项,笔者推荐你看一下由 Gergely Kontra 编写的 mu-template,可以到下面这个网址下载: http://www.vim.org/scripts/script.php?script_id=222.

4.1.3 snipMate 脚本

虽然已经有大量的模版系统脚本可供使用,而且它们中的大部分都是以本章介绍的概念为基础,但是有时候对于特定的文件格式, Vim 还是没有提供足够的支持. 对于这些情况,用户可以使用 snippet 脚本. Snippet 和用在模版中的缩写有点类似,但是要更高级一点.

如果读者想在 Vim 中使用 snippet, 笔者推荐 snipMate 脚本, 下载地址是 http://www.vim.org/scripts/script.php?script_id=2540.

snipMate 可以让用户在不需要了解 Vim 脚本的前提下, 为自己的文件格式定义高级的 snippet.

假设用户想为 for 循环创建一个 snippet (在前面一节里用的是 iabbrev), 那就这样做:

```
snippet for
    for (${1:i} = 0; $1 < ${2:count}; $1${3:++}) {
        ${4:/* code */}
    }
```

把上面这段文本写到一个文件中, 再把这个文件放到 \$VIMHOME/snippets 目录下, 把文件命名为 *file-type.snippet* (*filetype* 表示文件类型, 比如 C 代码就写成 *c.snippet*, PHP 代码就写成 *php.snippet*). 脚本的第一行表示开始一个新的 snippet, 当输入 *for* 并且后面再紧跟着一个 *Tab* 键时, 执行 snippet 脚本.

再接下来的一行是最神奇的, 脚本实际生成的代码看起来就像:

```
for (i = 0; I < count; i++) {
    /* code */
}
```

snippet 被插入之后, 它把光标放在了第 1 个 *i* 的位置, 并切换到插入模式, 可以轻易地把 *i* 换成另一个变量. 现在, 最神奇的事情发生了, 第 1 个 *i* 被换掉之后, 第 2 个 *i* (后面带有 ++ 的那个) 也会被自动更新成新变量. 修改了 *i* 之后 (如果需要的话), 按下 *Tab* 键, 光标就会跳转到 *count*, 并再次切换到插入模式. 现在你可以以同样的步骤修改 *count* 并按下 *Tab* 键, 之后, 光标跳转到 ++, 再按下 *Tab* 跳到 */* code */*, 你可以在这里写下其他代码.

在 snippet 的不同变量之间跳转, 简直不能再简单了.

snipMate 系统通过查找由 \${NUMBER:INITIAL_VALUE} 创建的特殊标记进行工作. 标记中的 NUMBER 指出了按下 *Tab* 时, 将要跳转到的序列. 如果 NUMBER 是 1, 那么光标会首先处在这个位置上, 如果 NUMBER 是 3, 那么它就是按下两次 *Tab* 后光标所在的位置, 以此类推. INITIAL_VALUE 是在修改 snippet 之前, 首先插入到 snippet 中的文本. 比如 \${1:i} 把 *i* 作为文本插入到 snippet 中, 并首先把光标放在此处.

如果需要在多个地方使用同一个变量, 可以通过 \$NUMBER 来引用. 比如说, 想要让 *i* 出现在多个地方, 那就在需要出现 *i* 的地方写上 \$1 即可. 写上 \$1 所有地方在开始时都会显示成 *i*, 当改变 *i* 时, 其他地方的 *i* 也会自动更新.

这里介绍的只是创建 snippet 的基础知识, 不过一旦在系统中安装了 snipMate, 就可以通过 Vim 的帮助系统学习到更多的高级知识.

还可以在 snipMate 中找到更多的惊喜. 用户会发现, snipMate 已经为许多常见的文件格式准备好了 snippet 脚本, 包括 .c, .php, .perl, .java, .html, .tex, 甚至还包括 Vim 脚本.



关于下载与如何使用 snipMate 脚本, 用户可以访问 http://www.vim.org/scripts/script.php?script_id=2540.

4.2 Tag List

Tag list 就像是程序员的字典. Tag list 实际上是一个包含了所有种类的关键词的文件, 这些关键词可以用来识别程序的各个要素, 包括函数名, 变量名, 类的方法, 具体的要素依赖于编程语言. Tag list 文件并非由 Vim 生成, 而是 tag list 生成程序. Tag list 生成程序有很多种, 比较常见的有:

- Exuberant Ctags: 用于 C, C++, Java, Perl, Python, Vim, Ruby (以及其他 25 种语言)
- Vtags: 用于 Verilog 文件
- Jtags: 用于 Java 文件
- Hdrtags: 用于 C/C++, Asm, Lex/Yacc, LaTeX, Vim, Maple
- Ptags: 用于 Perl 文件

因为 Exuberant Ctags (简称 Ctags) 使用得最为广泛, 支持的语言也是最多的, 因此在下面的例子里我们用它生成 tag list.

以一个小项目进行讲解, 这个项目包含了 3 个 C 文件:

- main.c: 包含 main 函数的文件
- myfunctions.c: 包含了程序中用到的各个函数
- myfunctions.h: myfunctions.c 中的函数的头文件

假设代码已经写完, 现在为这些代码文件生成一个 tag list 文件.

在存放源代码文件的目录下执行命令:

```
ctags *.c *.h
```

命令执行完毕后, 会在当前目录下创建一个新文件, 文件名是 tags. 这个文件就是 tag list 文件, 它包含了源代码中所有函数与变量的信息.



ctags 命令提供了大量的参数, 用于指定编程语言, 更多的信息可以通过 `ctags --help` 来查看.

启动 Vim 后, 必须告诉它去使用 tags 文件, 这可以通过设置选项 tags 来完成:

```
:set tags=/path/to/tags
```

现在, Vim 已经知道了 tags 文件的存在, 接下来就可以在 Vim 使用它.

main.c 调用了 myfunctions.c 中定义的函数. 假设用户知道有一个函数叫作 calcValue, 但却不知道该函数接收什么样的参数, 比较好的做法是看一下函数是如何定义的, 这时候就可以使用 tags 文件提供的功能. 假设用户已经在源代码文件中写下了:

```
myvalue = calcValue(
```

为了查看函数 calcValue 的定义, 把光标移动到函数名上, 按下组合键 **Ctrl+]**, 此时会出现下面两种情况中的一种:

- 只找到一个匹配, 于是光标会直接跳转到定义函数的地方
- 找到多个匹配, 于是 Vim 把这些匹配以列表的形式显示出来

在第 2 种情况下, 用户可以选择跳转到哪一个匹配. 如果用户使用的编程语言支持重载, 就会出现这种情况: 同一个函数有多个版本.

函数看完之后, 用户需要回到之前离开的地方, 继续往下工作. 按下组合键 **Ctrl+t** 就可以跳转到上一次离开的地方.



如果用的是 Gvim, 还可以通过鼠标来跳转到关键词的定义: 按住 *Ctrl* 键的同时, 按下鼠标左键。

用户可以把在 *tag* 间的跳转当成栈操作。当跳转到一个关键词时, 相当于把关键词的 *tag* 压栈, 从 *tag* 返回时, 相当于把栈顶的 *tag* 弹出栈。使用下面这个命令查看当前的栈状态:

82

```
:tags
```

```
js
TO tag          FROM line  in file/text
1 clear_edge_list  71  clear_edge_list();
1 edge_list       62  edge_list = NULL;
1 show_operation_failed_dialog  113  show_operation_failed_dialog();
as ENTER or type command to continue
```

上图中以 *>* 开始的行是当前所在的 *tag*。除了组合键 *Ctrl+]* 与 *Ctrl+t*, 还可以用下面这两个命令完成同样的功能:

- *:tag*: 跳转到这个 *tag*
- *:pop*: 回到上一个 *tag*

如果匹配的 *tag* 比较多, 要是能把这些 *tag* 以列表的形式罗列出来可能会比较方便。为了得到这个列表, 可以用下面两个命令中的任意一个:

- *:tselect*
- *:ptselect*

第 1 个命令列出所有匹配的 *tag*, 为了选择其中一个, 需要输入 *tag* 所在行的行首的数字。

第 2 个命令完成同样的工作, 但是它会在预览窗口中显示列表。如果用户选错了 *tag*, 或者是想查看列表中的下一个 *tag*, 可以用下面的命令在 *tag* 间移动:

- *:tnext*: 移动到列表中的下一个 *tag*
- *:tprev*: 移动到列表中的前一个 *tag*

用户可能还没有从前面的例子中体会到 *tag list* 的强大之外, 想像一下, 假设项目中的文件不只 3 个, 而是散布在数百个目录中的上千个文件, 这时候, 用户肯定没办法记住每个函数所在的位置, 这时候就非常需要一个强大的索引工具, 就像 *tag list*。

83

4.2.1 更便捷的 taglist 导航

在大多数的非英文键盘布局中, 按键 *]* 无法直接使用, 必须通过其他的组合键 — 比如 *Ctrl+Alt+G+r+9* — 完成 *Ctrl+]* 的功能。在这种情况下, 把命令映射到直接可用的按键上会更方便一点, 对此, 笔者使用了下面的映射:

```
:nmap <buffer> <F7> <C-]>
:nmap <buffer> <C-F7> <C-T>
:nmap <buffer> <A-F7> :ptselect<cr>
:nmap <buffer> <F8> :tnext<cr>
:nmap <buffer> <C-F8> :tprev<cr>
```

现在, 用户可以用 *F7* 与 *Ctrl+F7* 在 *tag* 间跳转, 用 *Alt+F7* 获取 *tag* 列表, 用 *F8* 与 *Ctrl+F8* 遍历 *tag*。

4.2.2 taglist 的其他用法

Taglist 不仅可以用来查找函数与变量的定义,还可以用来查找其他信息.笔者在这里只是简单地提一下 taglist 在 Vim 中的其他应用:

- `lookupfile.vim`: 由 Hari Krishna Dara 开发, 该脚本使用 `tag list` 在一个预处理过的项目中查找指定的文件, 脚本的最新版见 http://www.vim.org/scripts/script.php?script_id=1581.
- `taglist.vim`: 由 Yegappan Lakshmanan 开发的插件, 在程序员中非常流行. 它是一个完整的源代码浏览工具, 在一个分割的窗口中显示函数, 关键词, 变量, 定义的概览. 插件的下载地址是 http://www.vim.org/scripts/script.php?script_id=273.
- `ctags.vim`: 由 Gary Johnson 与 Alexey Marinichev 开发. 脚本可以在状态条或窗口标题栏中显示光标所在位置的函数的名字. 脚本使用 `Exuberant Ctags` 为当前打开的文件自动生成 `tag` 文件. 脚本的下载地址是 http://www.vim.org/scripts/script.php?script_id=610.
- `autoprotovim.vim`: Jochen Baier 专门为 C 程序员开发的脚本. 当程序员输入函数名与第一个左括号后, 脚本在预览窗口中显示该函数的原型. 脚本的下载地址是 http://www.vim.org/scripts/script.php?script_id=1553.

84



关于 `tag` 的更多信息与使用方法, 可以查阅 Vim 的帮助系统 `:help tags`.

4.3 使用自动补全

作为一个遵循 Vim 哲学的用户, 总是希望用最少的击键来完成一件事, 因为额外的击键意味着额外的时间.

所以说, 如果 Vim 可以猜出用户正想打的单词, 并且可以自动补全, 那又何必每次都从头输到尾呢?

在 Vim 中有多种办法来自动补全单词, 其中一些只能补全用户曾经在某个打开的缓冲区中输过的单词, 还有一些牵涉到了对当前正在使用的代码的分析 — 不仅仅是当前打开文件, 还包括整个源代码树.

接下来的小节讨论使用自动补全的三种方法:

- 已知单词的自动补全
- 基于字典文件的补全
- 识别上下文的自动补全

除此之外, 还会介绍一些使用技巧, 通过按键绑定来更方便地使用自动补全.

4.3.1 已知单词的自动补全

这一节将会看到自动补全的最简单的用法 — 为已知单词进行补全.

无论用户在写什么内容, 都会出现输入重复单词的情况. 在 Vim 中, 用户可以在输入完单词的头两个字母后, 按下组合键 `Ctrl+n`.

假设用户想要在 Vim 中输入 “I have beautiful flowers in my flower garden”.

85

在刚开始时, 文件中没有任何其他的内容, 因此用户不得不一个字母一个字母地写下文本的开始部分, 直到 “I have beautiful flowers in my f”.

用户接下来会输入单词 “flower”, 但是, 由于在前面已经输入了 “flowers”, 当输入 “f” 后, 按下 *Ctrl+n*, Vim 就会把 “f” 自动扩展为 “flowers”, 只要把多余的 “s” 删掉即可. 和输入整个单词相比, 这要快很多.

随着文本的增多, 读者会发现越来越多的单词可以自动补全.

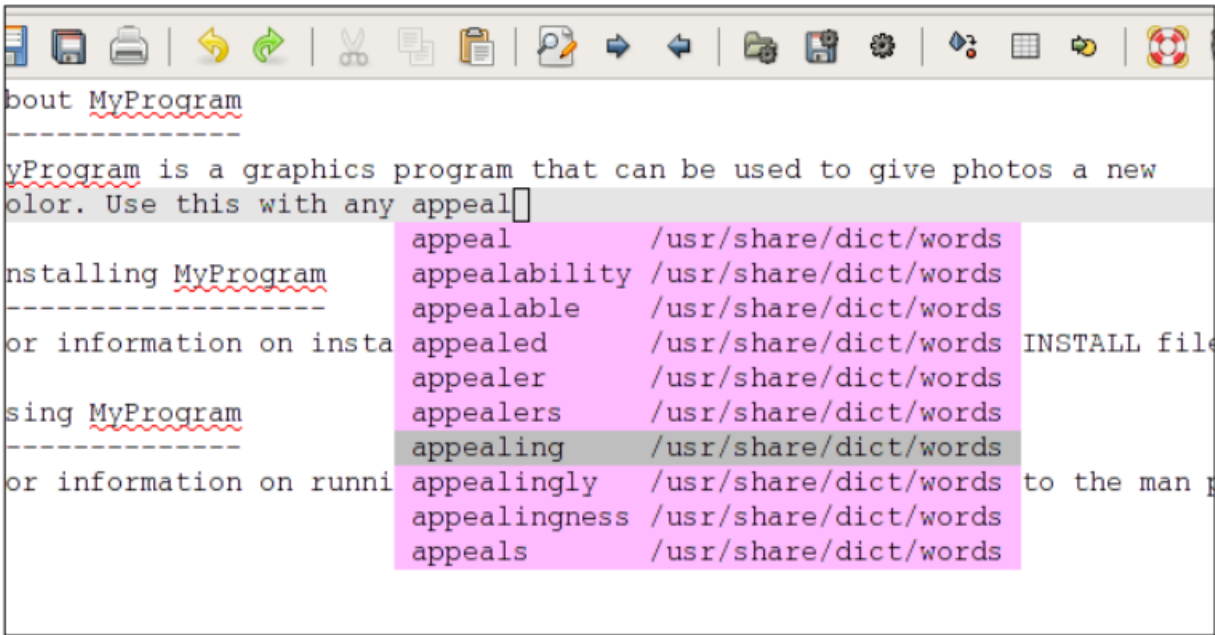
组合键 *Ctrl+n* 所做的工作是向前搜索匹配的单词. 如果用户知道自己刚刚才输入过同样的单词, 那么用 *Ctrl+p* 会更快一点, 因为这个组合键是向后搜索匹配. 一般来说, 除非文件非常大, 或者可能的匹配非常多, 否则 *Ctrl+n* 与 *Ctrl+p* 对用户来说并没有太大的区别.

4.3.2 使用字典的自动补全

一个比较好的使用技巧是把用户所使用的语言的全部单词, 都搜集到一个巨大的字典文件中, 然后再把这个文件作为字典加载到 Vim 中 (在因特网上很容易搜索到这种文件). 为了把文件作为字典载入到 Vim 中, 执行下面的命令:

```
:set dictionary+=/path/to/dictionary/file/with/words
```

现在, Vim 已经事先知晓了大量的单词, 用户可以利用这些单词进行自动补全. 不过, 有一些地方会不一样. 因为补全时被搜索的单词并非在某个打开的缓冲区中, 而是在选项 `dictionary` 所配置的字典文件中, 所以需要用另一个快捷键: *Ctrl+x+k*. 按下 *Ctrl+x* 后, 会进入补全模式, 再接着按下 *Ctrl+k*, 就可以在字典中查询关键词 (*k* 指的是 keyword).



还有其他几种补全类型, 包括:

Ctrl+x 后跟:

- *Ctrl+l*: 补全一整行文本
- *Ctrl+n*: 从当前缓冲区中补全单词
- *Ctrl+k*: 从字典中补全单词

- *Ctrl+t*: 从同义词典中补全单词 (见 `:help 'thesaurus'`)
- *Ctrl+i*: 从当前文件与被包含的文件中补全单词
- *S*: 拼写建议 (仅 Vim 7.0 以上的版本支持)

其他的将会在下节介绍.

4.3.3 omnicompletion

哪些内容应该自动补全, 哪些内容不应该自动补全 — 对此我们都有完美的解决办法. 但是对于 Vim 来说, 在 7.0 版出现之前, 用户对补全并没有绝对的控制权.

Vim 7.0 引入了一种全新的补全技术 — **omnicompletion**. 它使得用户可以精确地定义补全功能应该如何工作. 实际上, 用户需要自己来编写补全函数 (除非已经有人把函数写好了). 和前一节介绍的一样, 激活补全的方法是先输入几个字母, 按下 *Ctrl+x* 进入到补全模式, 再紧接着按下 *Ctrl+o* 使用 **omnicompletion**.

87

为了添加用户自定义的补全函数, 执行:

```
:set omnifunc=MyCompleteFunction
```

现在, 用户只需要定义函数 `MyCompleteFunction`, 这个函数用于完成补全操作. 上面的命令只在当前活动缓冲区内才有效, 如果希望每个缓冲区都可以使用该补全函数, 那么就要对每个缓冲区执行这个设置命令.



`omnifunc` 的设置通常在文件类型插件内完成, 这样的话, 补全函数就可以绑定到特定的文件类型上.

现在来看一个补全函数的例子. 假设用户有一个通讯录文件, 文件的内容是人名及其邮件地址, 就像:

```
Kim Schulz|kim@schulz.dk
John Doe|john.doe@somedomain.com
Jane Dame|jd@somedomain2.com
Johannes Burg|jobu@somedomain3.net
Kimberly B. Schwartz|kbs@somedomain.com
```

用户想在写完一个人名后, 通过补全插入对应的邮件地址, 完成这个功能的函数是:

```
function! CompleteEmails(findstart, base)
    if a:findstart
        " locate the start of the word
        let line = getline('.')
        let start = col('.') - 1
        while start > 0 && line[start - 1] =~ '\a'
            let start -= 1
        endwhile
        return start
    else
        " find contact names matching with "a:base"
        let res = []
```

```

" we read contactlist file and sort the result
for m in sort(readfile('/home/kim/.vim/contacts.txt'))
  if m =~ '^' . a:base
    let contactinfo = split(m, '|')

    " show names in list, but insert email address
    call add(res, {'word': contactinfo[1],
                  \ 'abbr': contactinfo[0].' <'.contactinfo[1].'>',
                  \ 'icase': 1} )
  endif
endfor
return res
endif
endfunction

```

88

函数接收两个参数 — 所有的 **omnifunction** 函数都是如此。Vim 第一次调用函数时, 把第 1 个参数 **findstart** 设置为 1 (**base** 为空)。这样的参数状态表示这是第一次调用, 函数应该搜索用户当前所写的单词的开始。

然后, Vim 再次调用函数, 这次调用把 **findstart** 设置为 0, 把 **base** 设置成开始补全的单词。这一次, 函数会打开通讯录文件, 以行为单位, 把文件读到一个列表中, 对列表排序, 然后遍历列表。

每一行的内容按照 `|` 分割, 如果某个单词, 是以用于补全的单词的字母作为开始, 那就把这个单词添加到结果列表中, 这个结果列表最后会作为函数的返回值返回。函数可以修改弹出内容的外观, 和它所匹配的内容, 这两个功能无法通过添加电子邮件地址来完成, 而是要通过构建一个字典 (见 `:help Dictionary`), 在构建字典时会接触到一些特定的关键词。在这个案例中使用下列三个关键词:

- **Word**: 应该插入的真正的单词
- **Abbr**: 在弹出列表中使用的单词, 而非直接使用 **word**
- **Icase**: 如果其值非零, 则匹配是区分大小写的

其他的关键词及其意义可以在 Vim 的帮助系统中找到:

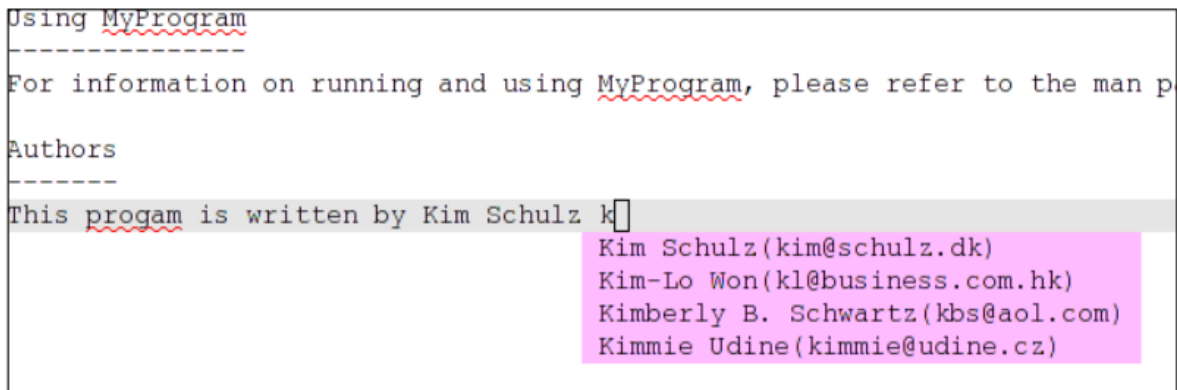
```
:help 'omnifunc'
```

现在 Vim 已经有了一个单词列表用于补全, 在这个案例中则是一些像下面这样的行:

```
"Kim Schulz <kim@schulz.dk>"
```

无论何时写下一些字母, 比如 `ki`, 然后按下 `Ctrl+x Ctrl+o`, Vim 就会以列表的形式, 弹出所有的, 以 `ki` 开始的名字。

89



按 `Ctrl+o` 可以循环遍历列表中的条目. 另外, 还可以用 `Ctrl+n` 向前遍历, 用 `Ctrl+p` 向后遍历.

4.3.4 多合一补全

用户可能不想记住这么多的补全快捷键, 为什么就不能为所有的补全类型只使用一个快捷键呢? Vim 可以帮助你实现这个愿望, 现在来看一下具体要怎么做.

大部分支持补全的编辑器通常把这个功能映射到 `Tab` 键.

通过下面这个命令, 用户可以在 Vim 的帮助系统中找到一个叫作 `CleverTab()` 的函数:

```
:help ins-completion
```

这个函数可以让用户使用 `Tab` 键来补全单词, 而非 `Ctrl+n`. 函数可以自动区分是应该插入一个制表符, 还是应该做补全操作. 如果用户是在一行的开始, 或者是某个空白符的后面按下 `Tab` 键, 那就是插入一个制表符. 在其他的情况下, 函数会试图完成已知单词的补全.

借用帮助系统中的 `CleverTab` 函数, 并对它加以扩展, 使得它可以从下面的, 带有优先级的列表中选择补全方法:

- omnicompletion
- 字典补全
- 已知单词的补全

下面是实现该函数的一个例子:

```
function! SuperCleverTab()
    'check if at beginning of line or after a space
    if strpart( getline('.'), 0, col('.')-1 ) =~ '\s*$'
        return "\<Tab>"
    else
        " do we have omni completion available
        if &omnifunc != ''
            " use omni-completion 1. priority
            return "\<C-X>\<C-O>"
        elseif &dictionary != ''
            " no omni completion, try dictionary completio
```

```

        return "\<C-K>"
    else
        "use omni completion or dictionary completion
        "use known-word completion
        return "\<C-N>"
    endif
endif
endfunction

" bind function to the tab key
inoremap <Tab> <C-R>=SuperCleverTab()<cr>

```

把函数的代码与按键绑定添加到 `vimrc` 中, 然后用户就可以用 *Tab* 键来完成各种类型的补全。

按下 *Tab* 键进行补全时, 函数检查此时是否是要插入一个制表符。如果不是, 那就检查 `omnicompletion` 函数 (通过 `omnifunc` 设置) 是否可用。如果函数不可用, 那就检查字典是否可用。如果字典还是不可用, 那就使用最简单的已知单词补全。

4.4 宏录制

编辑具有单一结构的文本时, 或许最好的功能就是录制一段输入宏, 然后再重复执行宏。

完成这个工作的接口非常简单, 更强大的是, 几乎任何东西都可以录制, 接下来我们就来介绍这个强有力的工具。

91

先看一下将会用到的命令:

- `qa`: 从现在开始录制, 并记录到寄存器 `a`。可以使用任意的寄存器, 用寄存器 `a` 只是为方便起见。
- `q`: 如果是在正在录制时按下, 则停止录制
- `@a`: 执行记录在寄存器 `a` 中的宏 (可以把 `a` 替换成其他的寄存器)
- `@@`: 执行上一次所执行的宏命令

可以在 `@` 前加上任意的数字, 表示重复执行该命令多少次。例如, `15@a` 表示把寄存器 `a` 中所记录的命令重复执行 15 遍。

先看一个普通的录制会话过程:

```

qq
command1
command2
....
commandN
q
10@q

```

读者可能会好奇这有什么用, 因为这只是在重复执行一串命令罢了。这种问题最好通过例子来说明。

想像一下用户有一大串的信息需要处理, `Unix` 系统的日志文件就属于这种类型。日志文件的内容类似于:

```
Mar 20 17:23:54 Inspiron kernel: [33604.866998] ath9k 0000:09:00.0: no
hotplug settings from platform
Mar 20 17:23:54 Inspiron kernel: [33604.867007] pcieport 0000:00:1c.7: no
hotplug settings from platform
Mar 20 17:23:54 Inspiron kernel: [33604.867015] ehci-pci 0000:00:1d.0: no
hotplug settings from platform
Mar 20 17:23:54 Inspiron kernel: [33604.867020] ehci-pci 0000:00:1d.0: using
default PCI settings
Mar 20 17:23:54 Inspiron kernel: [33604.867039] lpc_ich 0000:00:1f.0: no
hotplug settings from platform
Mar 20 17:23:54 Inspiron kernel: [33604.867043] lpc_ich 0000:00:1f.0: using
default PCI settings
Mar 20 17:23:54 Inspiron kernel: [33604.867063] ahci 0000:00:1f.2: no hotplug
settings from platform
Mar 20 17:23:54 Inspiron kernel: [33604.867067] ahci 0000:00:1f.2: using
default PCI settings
Mar 20 17:23:54 Inspiron kernel: [33604.867084] pci 0000:00:1f.3: no hotplug
settings from platform
Mar 20 17:23:54 Inspiron kernel: [33604.867089] pci 0000:00:1f.3: using
default PCI settings
...
```

现在, 用户想要把这个日志文件转换成 **HTML**, 并以表格的形式呈现数据, 看起来就像:

```
<tr><td>Oct 8 21:23:34</td><td>laptopia</td><td>kernel:</td><td>ACPI...</td><tr>
```

用户可以选择一行一行地编辑文件, 直到最后一行. 但也可以这样做, 先编辑一行, 并把编辑命令录制下来, 然后再把录制下来的命令应用到剩下的每一行. 最开始的编辑命令可以是下面这些 (假设光标原来是在第一行的开始):

qa	开始录制, 并把内容记录到寄存器 a 中
i<tr><td>[ESC]	切换到插入模式, 插入 HTML 标记, 再回到普通模式
/ [CR]	向前搜索空格
3n	向前搜索到第 3 个空格
xi</td><td>[ESC]	删除空格, 切换到插入模式, 添加 HTML 标记, 再切换到普通模式
n	前进到下一个空格
xi</td><td>[ESC]	删除空格, 切换到插入模式, 添加 HTML 标记, 再切换到普通模式
n	前进到下一个空格
xi</td><td>[ESC]	删除空格, 切换到插入模式, 添加 HTML 标记, 再切换到普通模式
A</td></tr>[ESC]	在行的末尾添加 HTML 标记, 再切换到普通模式
j	前进到下一行的开始
q	宏录制结束



[ESC] 指的是按下转码键: *Esc*, [CR] 指的是按下回车键: *Enter*.

宏准备好了, 光标的位置也已就位, 现在可以重复地执行宏, 并且在每次执行完宏后, 光标都会处在就绪的位置上.

用户可以用 @a 回放存放在寄存器 a 中的宏, 除此之外, 用户还需要做的就是添加 **HTML** 的头部与尾部信息, 这很简单. 这只是宏应用的一个简单示例, 如果读者认真回忆一下, 可能会想到更多的, 可以用宏来优化的工作.

93

4.5 使用会话

不知读者有没有想过, Vim 会为你保存多少信息. Vim 保存的信息涵盖了大量的内容, 其中包括:

- 打开的文件, 缓冲区, 窗口, 和标签页
- 历史命令
- 文本的变化点
- 选择与撤消分支
- 窗口, 分割与 GUI 窗口的大小
- 光标的位置

保存的信息可以分为三类:

- 第一类设置信息称为视图 (**view**), 该类信息应用到一个单独的窗口上. 一个视图可以被存储和还原, 因此每当用户使用视图时, 窗口都会呈现出相同的外观.
- 第二类设置信息称为会话 (**session**). 它是一系列视图的集合, 再加上视图之间如何配合的信息. 和视图一样, 会话也可以保存下来, 以便于稍后检索.
- 剩下的其他信息都归为第三类, 也就是所有的, 不能直接应用到任意一个窗口的全局设置. 这些设置可以用会话保存下来, 因此它们也可以保存/还原.

接下来的小节将会介绍如何在日常的工作中运用会话.

4.5.1 简单的会话使用

在使用会话的过程中, 最经常做的事就是把当前正在运行的会话 (如果没有经过特别的设置, 则是默认会话) 保存到会话文件中, 这样就可以在需要时重新加载会话. 保存会话的命令是:

```
:mksession file
```

保存当前视图的命令是:

```
:mkview file
```

命令中的 *file* 是用户指定的, 用于保存会话或视图的文件名. 如果没有指定该参数, 则默认保存到目前工作目录的 `Session.vim` 文件中.



如果文件事先存在, 为了覆盖掉文件原来的内容, 需要在 `mksession` 的末尾紧跟上一个感叹号 `!`.

使用视图时, 用户可以同时拥有多个不同的视图. 如果每个视图都保存到目前工作目录下, 那么用不了多长时间, 工作目录就会被视图文件填满. 为了避免这个问题, 可以用下面的命令告诉 **Vim**, 应该把视图文件放到哪个目录下:

```
:set viewdir=$HOME/.vim/views
```

上面的命令会把所有的视图文件保存到目录 `$HOME/.vim/views`.

假设用户当前打开了三个窗口, 在退出 **Vim** 之前, 用户执行了:

```
:mksession
```

在下次准备打开文件时, 用户希望用相同的会话来启动 **Vim**, 此时可以用命令行选项 `-s`:

```
vim -S Session.vim
```

除此之外, 用户也可以选择启动 **Vim** 后, 用下面的命令来加载会话文件:

```
:source Session.vim
```

对于视图, 加载方式是:

```
:loadview View.vim
```

加载一个会话会改变编辑器的整体布局, 而加载一个视图则只会改变当前活动窗口的布局.

用户如果希望 **Vim** 记住所有的设置信息, 比如光标的位置与折叠信息, 那么就需要在 `vimrc` 中添加如下内容:



```
set viewdir=$VIMHOME/views/
autocmd BufWinLeave * mkview
autocmd BufWinEnter * silent loadview
```

无论何时在同一个窗口中打开另一个缓冲区, 之前的缓冲区的视图都会被自动保存下来, 并且当再次打开之前的缓冲区时, 会自动还原保存的视图.

会话的一个常用技巧是定义一个命令, 命令的功能是在退出 **Vim** 时自动保存会话, 而在打开 **Vim** 时自动还原保存的会话. 使用这个方法就可以让用户在不丢失设置, 已打开文件列表等信息的前提下, 自由地启动与退出 **Vim**. 为了完成这个功能, 可以在 `vimrc` 中添加如下命令:

```
autocmd VimEnter * call LoadSession()
autocmd VimLeave * call SaveSession()
function! SaveSession()
    execute 'mksession! $HOME/.vim/sessions/session.vim'
endfunction
```

```
function! LoadSession()  
    if argc() == 0  
        execute 'source $HOME/.vim/sessions/session.vim'  
    endif  
endfunction
```

关闭 Vim 后, 会话保存到 `$HOME/.vim/sessions/session.vim`.

根据打开 Vim 方式的不同, 或者是打开由命令行参数指定的文件, 或者是打开最后一次会话. 比如说:

- `vim file.txt`: 这种打开方式不会加载最后一次会话.
- `vim`: 这种打开方式会加载最后一次会话, 之前打开的文件会再次被打开.

如果用户希望在会话文件中存放更多的信息, 可以把额外的信息存放到额外的会话文件中. 方法是创建一个其名字类似于会话文件的文件, 不过要把扩展名 `.vim` 改成 `x.vim`. 例如, `Session.vim` 的额外会话文件是 `Sessionx.vim`. 额外的会话文件应该和其所属的会话文件放在同一目录下. 用户可以把想要添加的命令全写到这个文件中, 当 Vim 加载会话文件时, 会把额外的会话文件中的命令执行一遍.

96

4.5.2 满足个人的会话需求

用户可能并不需要会话文件中存放的所有信息, 有时, 用户可能仅仅是想要保存打开过的文件的信息, 而在其他时候, 可能是所有的会话信息. 幸运的是, Vim 提供了设置保存会话信息的方法.

所使用的设置命令是 `sessionoptions`, 使用方法是:

```
:set sessionoptions=options
```

options 是一个由逗号分隔的列表, 列表中可以出现下列选项:

blank	保存空白窗口
buffers	保存所有缓冲区的信息, 包括隐藏的与未加载的缓冲区
curdir	保存当前工作目录的相关信息
fold	保存缓冲区内容中的折叠信息
globals	保存全局变量的相关信息, 这里的全局变量指的是以大写字母开始, 且类型为字符串或数值的变量
help	保存帮助窗口
localoptions	保存用户在单个窗口中创建的局部变量与局部映射的相关信息
options	保存所有的选项信息, 包含全局的与局部的
resize	保存 UI 窗口的尺寸信息 (行数与列数)
sesdir	如果设置了该选项, 当前目录就会是会话文件的存放目录 (如果同时设置了 curdir , 则该选项不可用)
slash	把所有文件路径中的反斜杆换成斜杆 (这样做就可以让 Unix 兼容 Windows 的文件路径)
tabpages	保存所有的标签页信息, 如果没有设置该选项, 则只会保存当前处于活跃状态的标签页
unix	使用 Unix 格式的行结束符, 而非 Windows 格式
winpos	保存 UI 窗口的屏幕位置信息
winsize	保存所有打开窗口的尺寸

粗体显示的是 Vim 的默认设置选项. 每次设置选项时并不需要全部设置, 可以通过运算符 `+=` 和 `-=` 来增加或移除选项. 比如说, 用户想要在默认选项的基础上添加 `winpos`, 并移除 `fold`, 可以这样做:

```
:set sessionoptions+=winpos
:set sessionoptions-=fold
```

用下面的命令显示当前的会话选项:

```
:echo &sessionoptions
```



实际上, 用户可以用该方法查看所有的 Vim 设置: 通过执行命令 `:echo`, 并在选项名的前面加上 `&`, 比如 `:echo &some sessting`.

4.5.3 会话与项目管理

有时候, 用户可能想要把会话文件作为某种原始的项目文件来使用, 这些项目文件包含了项目的相关信息. 因此, 如果用户正在处理某个项目, 并且打开了大量的文件与窗口, 只需执行:

```
:mksession!
```

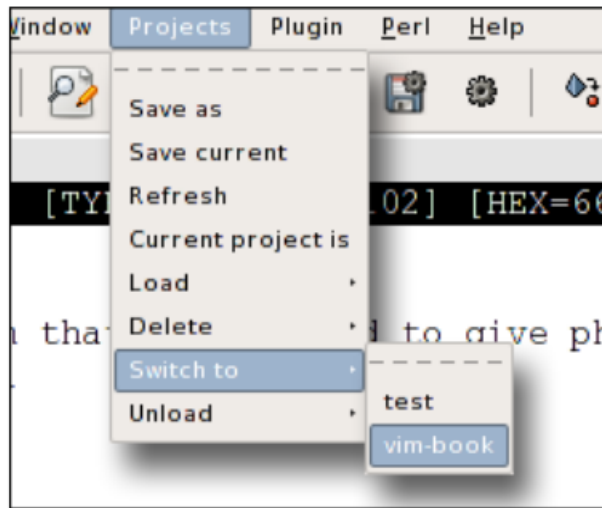
该命令会把当前会话保存到当前工作目录的 `Session.vim` 文件中. 如果用户希望在启动 Vim 时, 自动加载会话文件 (如果存在的话), 可以把下面这行命令添加到 `vimrc`:

```
silent source! Session.vim
```

添加了该命令之后, 如果在当前工作目录中存在 `Session.vim` 文件, 就可以自动加载. 所以, 只要让会话文件保持在项目目录中, 用户就可以方便把项目加载到 Vim, 但是请记住, 如果打开了新文件, 或者是修改了窗口/缓冲区, 都需要重新保存会话文件.

98

这只是把会话应用到项目管理的简单用法, 当然, 还有更高级的方法. Wenzhi Liang 开发了一个实用的脚本, 在 Gvim 的菜单中添加了一个 **Project** 菜单项. 通过这个菜单项, 用户可以把当前会话保存为以项目命名的文件, 之后, 再通过点击菜单来还原项目, 还可以在不同的项目之间切换. 如果用户不再需要某个项目, 还可以通过菜单来删除该项目的会话文件.



关于该脚本的更多信息, 以及下载地址, 见: http://www.vim.org/scripts/script.php?script_id=279



该脚本要求系统中安装了 Perl 与 Bash.

4.6 寄存器与撤消分支

用户可能都有过这种体验: 刚刚才删除完一段文本, 可是过了一会儿发现, 其他地方还需要那段文本; 或者是已经复制/剪切了一段文本, 但是过了一会儿之后, 之前复制/剪切的文本就不见了, 因为剪贴板只能存放一段文本, 后面的文本会把前面的文本给覆盖掉 — Vim 是不会让这种事情发生的.

当用户修改文本, 并且需要跟踪被删除/复制的文本, 又或者是跟踪通常情况下对文本作出的修改时, 可以使用 Vim 提供的两个工具:

- 寄存器: 寄存器是一种带有多个缓冲区的高级剪贴板, 可以用来存放被剪切, 删除或复制的文本.
- 撤消分支: 撤消分支是 Vim 版本控制的一种简单形式. 利用撤消分支, 用户可以把文件回滚到某个特定的时刻, 或者是回滚一定的次数. 如果用户对某些撤消操作感到后悔, 还可以回过头来找到包含这些修改的撤消分支.

99

接下来的两节将会向读者介绍如何在日常的工作中运用寄存器与撤消分支. 读完之后, 读者将会认识到它们的强大之外, 以及它们在日常工作是如何帮助用户.

4.6.1 使用寄存器

在许多程序与操作系统中, 用户只能利用一个剪贴板来完成文本的剪切与复制. 在 Vim 中不会出现这种情况, 因为用户可以访问多达 9 种不同的剪贴板, 更准确来说, 应该是 9 种不同的寄存器类型.

有些寄存器类型的工作领域有所重叠, 但还有一些则拥有自己独特的目的. 用户可以在多种不同的命令或操作中 (比如复制, 删除, 粘贴) 使用寄存器. 寄存器的名字都以双引号 " 开始, 比如 "x, 所以, 现在来看一下如何使用一个寄存器. 假设现在要用的寄存器是 "x, 至于 x 实际上是表示哪个寄存器, 稍后解释.

为了把某段文本复制到寄存器中, 可以用命令 y 来完成, 不过这次有所不同, 在复制之前, 先告诉 Vim 被复制文本的存放位置:

```
"xy (用 "xyy 可以复制一整行)
```

对于用来剪切文本的命令 x 也是一样的:

```
"xx
```

对于删除文本的命令 d 则是:

```
"xd
```

现在, 用户已经在寄存器 "x 中存放了一段文本, 并且想要把这段文本粘贴到正式文本中, 可以用命令 P (粘贴在光标位置之前) 或 p (粘贴在光标位置之后), 但在粘贴之前, 要先指定寄存器:

```
"xP 或 "xp
```

如果用户忘记自己使用了哪些寄存器, 可以执行命令:

```
:registers
```

现在, 用户已经知道了如何在基本命令中使用寄存器, 那么接下来将会分别介绍每一种寄存器类型.

匿名寄存器

之所以称为匿名寄存器, 是因为它只能通过 "" 来访问. 每当用户使用命令 y 复制文本, 或者用 d (删除), c (删除, 并进入插入模式), s (替换), x (剪切) 删除文本时, Vim 都会自动填充该寄存器. 匿名寄存器始终指向最近使用的寄存器, 也就是说, 即使用户在删除/复制文本时, 指定了一个特定的寄存器, 匿名寄存器也可以正常工作. 比如说, 命令 "xdd 会把被删除的文本同时存入寄存器 x 与匿名寄存器.

如果用命令 p 或 P 粘贴文本时没有指定寄存器, 那么 Vim 就从匿名寄存器获取被粘贴的文本.

行内删除寄存器

如果用户删除的文本小于一行, 那么 Vim 就把删除的文本存入一个特殊的寄存器 — 行内删除寄存器 "-, 前提是用户没有指定其他的寄存器.

带编号的寄存器

带编号的寄存器包括 "0, "1, 一直到 "9. 带编号的寄存器分为两类, 第一类是寄存器 "0, 它总是包含了最近一次被删除 (d 或 x)/修改 (c) 的文本. 如果用户删除或修改了新的文本, 寄存器 "0 的内容就会被新文本覆盖.

类似于寄存器 "0, 寄存器 "1 也包含了最近一次被删除/修改的文本. 然而, 如果用户指定了其他的寄存器, 又或者是文本的长度少于一行 (此时会自动使用行内删除寄存器), 那么寄存器 "1 的内容就不会被覆盖. 为了和 vi 兼容, 如果在删除/修改文本时, 使用下面这些移动命令: %, (,), {, }, `, /, ?, n, N, 那就总是使用寄存器 "1, 如果文本少于一行, 还会把文本存入寄存器 "-.

101

和寄存器 "0 有所不同的是, 寄存器 "1 的内容并不会因为新内容的加入而被删除, 相反, 原有的内容会移到寄存器 "2. 如果寄存器 "2 原来是非空的, 那么寄存器 "2 的内容会先移到寄存器 "3, 以此类推, 直到寄存器 "9. 寄存器 "9 的原有内容会由于新内容的加入而被覆盖掉. 通过这种方式, 寄存器 "1 到寄存器 "9 就可以记住删除/修改的历史内容, 即使用户删除了新的文本, 也可以通过这 9 个寄存器访问到早先被删除的文本.

命名寄存器

命名寄存器分为两种 — "a 到 "z 与 "A 到 "Z.

如果使用的是小写字母的寄存器, 比如 "a, 那么它们就像普通的寄存器那样工作 — 将删除或复制的文本存入该寄存器. 当有新的内容被添加进来时, 寄存器中原来的内容就会被覆盖掉.

如果使用的是大写字母的寄存器, 比如 "A, 那么当有新的内容被添加进来时, 原来的内容并不会被覆盖掉, 而是把新内容追加到原有内容的末尾.



如果往选项 `coptions` 添加新值 '>', 那么追加到大写字母寄存器的新文本与旧文本之间将会以换行符分开, 给选项设置新值的命令是: `:set coptions+='>'`.

因为用户对命名寄存器具有完全的控制力, 因此用户应该最先熟悉如何使用它们.

只读寄存器

有四个只读寄存器, 它们的特殊之处在于只有 Vim 才有权限修改它们. 用户只能通过命令 `P, p` 或 `:put` 访问它们的内容. 四个只读寄存器存放的内容有所不同:

- "%: 该寄存器包含了当前活动缓冲区的文件名.
- "#: 该寄存器包含了当前活动缓冲区上一次打开的文件, 也被称为备选文件.
- ".: 该寄存器总是包含了最近一次插入的文本. 因此, 用户可以通过执行 `" .p` 来重复粘贴最近一次插入的文本.
- "::: 该寄存器包含了用户上一次在命令行执行的命令. 如果用户重复执行一条历史命令, 那么寄存器的内容就不会被覆盖. 为了使被执行的命令存入该寄存器, 用户至少需要输入命令中的一个字符.

102

选择与投递寄存器

这种寄存器类型包含了三个寄存器: "*", "+" 与 "~. 这 3 个寄存器用于存放与检索用户在 Gvim 中选择的文本. 寄存器 "*" 实际上访问的是窗口系统的剪贴板. 如果使用的操作系统是 Microsoft Windows, 那么用户在使用寄存器 "*" 与寄存器 "+" 并不会感到有什么不同. 然而, 如果是 Linux, 那就不一样了, 因为 X11 (Linux 的窗口系统) 的选择寄存器不止一个, 而是三个. 寄存器 "+" 存放的是用户所选择的任意一段文本, 典型的选择方式是通过鼠标完成. 然而, 仅当用户显式地告诉 Vim 去复制文本时, 寄存器 "*" 的内容才会改变.

任意一个 GUI 应用程序都可以访问这些寄存器, 它们的使用方法和日常的复制粘贴没什么区别.

最后一个寄存器 `"~` 称为投递寄存器, 它包含了最近一次投递到 Vim 的文本. 所以说, 如果用户在另一个程序中选择了一段文本, 并把它拖动到 Vim 的窗口中, 那么寄存器 `"~` 就包含了这段文本.

黑洞寄存器

顾名思义, 这个寄存器就像黑洞那样工作 — 进入该寄存器的任意内容都无法再被取回. 如果用户希望彻底删除某些文本, 也不想让任意一个寄存器记录被删除的文本, 此时就可以用黑洞寄存器. 黑洞寄存器的名字是 `"_`, 使用寄存器的例子有 `"_x` 或 `"_dd`. 如果用户试图读取刚刚写入到该寄存器的内容, 将会发现, 无论如何尝试, 都不会返回任何内容.

搜索模式寄存器

无论何时使用命令 `/`, 被搜索的模式都会自动存入搜索模式寄存器. 寄存器的名字 `" /` 非常容易记忆, 因为它就是在搜索命令 `/` 前加上了双引号, 表明这是一个寄存器. 如果打开了选项 `hlsearch`, Vim 就会根据寄存器中的内容来高亮文本. 用户可以利用这个特点, 修改寄存器的内容, 从而高亮显示其他文本. 在不执行搜索的前提下修改寄存器内容的方法是:^①

```
:let @/= "pattern"
```

命令中的 *pattern* 是 `hlsearch` 将要高亮显示的字符串.

表达式寄存器

Vim 的最后一种寄存器类型是表达式寄存器, 然而, 称它为寄存器并不恰当, 因为它并不像其他通常的寄存器那样存放文本, 用户甚至不能写该寄存器. 取而代之的是, 它允许用户访问命令行, 对表达式求值并返回运算结果, 就好像它一开始就存放在寄存器中似的.

访问表达式寄存器的方式是输入它的名字 `"=`. 输入完等号后, 光标自动跳转到命令行窗口, 如果命令行窗口的第一个字符是等号, 则说明用户现在工作在表达式寄存器. 现在用户可以输入待求值的表达式, 并以回车键结束, 之后, 可以用命令 `p` 或 `:put` 把运算结果粘贴到正文中. 如果不想对已输入的表达式进行求值, 则按 `Esc` 结束. 如果用户在按回车键之前没有输入表达式, Vim 就会使用上一次输入的表达式. 表达式必须是有效的, 而且应该返回一个字符串. 如果表达式的结果是数值, Vim 会自动把它转换成字符串. 如果用户无法确定运算结果的类型, 可以使用函数 `string()` 把结果转换成字符串中再返回.^②



执行 `:help expression`, 查看如何输入有效的表达式.

4.6.2 撤消分支

用户应该都执行过对已修改文本的撤消操作. 通常情况下, 执行撤消操作的方式是按工具栏的撤消按钮, 或者是组合键, 这样, 最后一次对文本的修改就会被撤消.

在这一方面, Vim 走得更远一些, 而且新增了对分支的支持.

这一节介绍什么是撤消分支, 以及如何在日常工作中使用它.

^①原文是 `:let "/ = pattern`, 应该是笔误. — 译者注

^②对表达式寄存器更方便的用法是: 在插入模式下, 按 `<c-r>=` (按组合键 `Ctrl+R`, 再按等号), 接着输入待求值的表达式, 输入完按回车键, 运算结果就会自动插入到正文中. — 译者注

先来介绍一下什么是撤消分支. 假设现在有一个文件, 而用户已经在它上面作了很多修改, 这时, 用户突然意识到最后四次更改是错误的. 通常来说, 这时用户会执行四次撤消命令 (或者直接执行 `4u`), 这样的话, 最后四次更改就会被取消. 在这里执行的撤消操作和其他编辑器中的撤消操作没什么不同. 假设现在用户想要对文件作另一个更改, 比如说改正一个拼写错误.

如果用户此时修改了拼写错误, 那么, 一般来说, 之前被撤消的四次更改操作, 其信息都会被丢掉 — 不过 **Vim** 并不会这样做.

当用户撤消了四次修改, 并添加了一个新修改时, **Vim** 会在撤消分支树上添加一条新分支, 此时的撤消分支树看起来就像:

```

      A-B-C-D
      /
    E-F-G-H
  
```

其中一条分支包含了四个已经被撤消的修改, 另一条分支则包含了最近几次的修改 (拼写改正). 如果用户继续编辑文本 (不包含撤消操作), 那么撤消树中就只会这两条分支. 如果用户在撤消某些操作后, 又作了新的修改, 撤消树就会新增一条分支. 到了最后, 用户将会看到一棵长满分支的撤消树, 它包含了所有的撤消与编辑操作.

下面这个命令可以查看当前存在着的撤消分支的概览:

```
:undolist
```

命令会显示分支的三种信息 — 修改号 (用于标识一条分支), 分支所包含的修改的次数, 分支上最后一次修改发生的时间, 命令的输出信息就像:

```

number  changes  time ~
6        5        12:12:11
11       8        14:01:15
  
```

如果用户想要切换到某个特定的修改号, 执行:

```
:undo n
```

命令中的 *n* 是修改号.

可以用下面的命令在修改列表中后退:

`g-`(如果是前进的话, 则是 `g+`)

那么, `g-` 与 `u` 之间有什么区别? 让我们通过例子来说明.

在 **Vim** 的编辑区中输入下面的文本:

```
My name is Jim
```

把光标移动到跳到字母 `J`, 按三次 `x` 来删除名字 `Jim`, 文本变成了:

```

My name is Jim
My name is im
My name is m
My name is
  
```

现在, 用户突然意识到正确的名字是 `Jimmy`, 于是他撤消了修改:

```
My name is m
My name is im
My name is Jim
```

撤消之后, Vim 中已经有了一条删除名字 Jim 的撤消分支. 现在把名字改为 Jimmy:

```
My name is Jimm
My name is Jimmy
```

但是, 用户真正的名字其实是 Kim, 而非 Jimmy. 因为 Jim 已经和 Kim 很接近了, 所以用命令 u 回滚, 并把 J 改成 K:

```
My name is Jimm
My name is Jim
My name is im
My name is Kim
```

把 Jim 回滚到 Jimmy 后, 会新增一条撤消分支.

现在, 用命令 g- 在修改列表中后退:

```
My name is Kim
My name is im
```

(从这里开始, Vim 切换到一条新的分支)

```
My name is Jim
My name is Jimm
My name is Jimmy
```

(从这里开始, Vim 切换到一条新的分支)

```
My name is
My name is m
My name is im
My name is Jim
```

如果用的是命令 u, 则变化过程是:

```
My name is Kim
My name is im
My name is Jim
My name is Jimm
My name is Jimmy
My name is Jim
```

可以看到, 命令 u 只是单纯地撤消施加上文本上的修改, 并不理会分支的变化. 而命令 g- 则会按照分支来进行撤消.

简单来说, 撤消分支使得用户可以访问到文本曾经经历过的任何状态.

除了在分支中一步一步地回退之外, 还可以直接跳转到某个时间点的文本状态. 为了完成这个功能, Vim 提供了两个命令, 分别用来在撤消历史中向前跳转和向后跳转, 命令的执行形式是:

```
:earlier Ns
:earlier Nm
:earlier Nh
:later Ns
:later Nm
:later Nh
```

命令中的 *N* 表示跳过的秒数 (s), 分种数 (m), 或小时数 (h). 命令 `:undolist` 可以显示分支上最后一次修改发生的时间, 通过这个信息, 用户可以大致计算出需要跳过的时间长度. 熟悉撤消分支的使用可能会需要点时间, 可是一旦上手之后, 它会对用户的日常工作产生极大的帮助.

107

4.7 折叠

一般来说, 编辑大型文件时 (尤其是源代码文件), 用户很难获取一份好的概览. Vim 提供了一个特性可以解决这个问题 — 折叠文本块. 这一节介绍如何利用折叠获取文本内容的概览.

折叠指的是将一个范围内的行 (比如一个函数的定义) 折叠成一行, 但不丢失文本内容. 比如折叠下面的文本:

```
function myFunction() {
    var a = 1;
    var b = 0;
    var c = a + b;
    return c;
}
```

折叠后的效果类似于:

```
+-- 6 lines: function myFunction() {
```

在这个示例中, 折叠是根据代码的语法, 用花括号来判断被折叠的范围. Vim 可以根据下面的信息来折叠文本:

- 手动折叠: 由用户手动标出被折叠的范围 (见 `:help fold-manual`)
- 缩进折叠: 根据缩进来折叠文本 (见 `:help fold-indent`)
- 表达式折叠: 根据表达式来折叠文本 (见 `:help fold-expr`)
- 语法折叠: 根据语法来折叠文本 (见 `:help fold-syntax`)
- 差异折叠: 折叠未被修改的文本 (见 `:help fold-diff`)
- 标记折叠: 根据文本中插入的标记来折叠文本 (见 `:help fold-marker`)

具体使用哪一种指示信息取决于文本的类型, 以及用户的具体操作.

108

现在开始介绍如何完成折叠操作, 在这之前, 要做的第一件事是打开折叠选项:

```
:set foldenable
```

打开后, Vim 就会注意到在普通模式下输入的折叠命令. 用来打开和关闭折叠的命令有很多个, 主要的有:

- zc: 关闭一个折叠
- zo: 打开一个折叠
- zM: 关闭所有的折叠
- zR: 打开所有的折叠

假设把语法折叠作为折叠时的指示信息, 首先把光标移动到需要折叠的区域 (比如某个函数的内部), 切换到普通模式, 执行命令 `zc` 来关闭折叠, 现在, 用户就可以看到函数的定义代码被折叠成一行. 下面的图片显示了折叠与未折叠的代码:

```

79 void show_save_graph_as_dialog( GtkMenuItem* menu_item, gpointer user_data ){
80     char* filename_selected;
81     gint response;
82     GtkWidget* save_graph_as_dialog = glade_xml_get_widget( xml, "save_graph_as_dialog" );
83     response = gtk_dialog_run( (GtkDialog*) save_graph_as_dialog );
84     gtk_widget_hide( save_graph_as_dialog );
85
86 -- if( GTK_RESPONSE_OK == response ){...6 Lines... }
92 }
93
94 void show_save_rendering_dialog( GtkWidget* widget, gpointer user_data ){...16 Lines...}
110
111 void show_render_window( GtkWidget* widget, gpointer user_data ){
112     GtkWidget* render_win;
113     GtkWidget* render_drawing_area;

```



如果用户不想记住打开与关闭折叠的命令, 可以把打开或关闭折叠的切换命令绑定到一个按键上, 比如空格: `:nnoremap <space> za`

如果用户觉得折叠后的行无法提供自己想要的信息, 可以通过修改选项 `foldtext` 的值来改变折叠行所显示的信息. 修改的方式是让 `foldtext` 指向另一个函数, 这个函数返回用户希望看到的信息:

```
:set foldtext=MyFoldFunction()
```

函数 `MyFoldFunction()` 的实现代码是:

109

```

function! MyFoldFunction()
    let line = getline(v:foldstart)
    " cleanup unwanted things in first line
    let sub = substitute(line, '/\*\|\/\*\|\/\^\s+', '', 'g')
    " calculate lines in folded text
    let lines = v:foldend - v:foldstart + 1
    return v:folddashes.sub.'...'.lines.' Lines...'.getline(v:foldend)
endfunction

```

函数使得折叠后的行变成:

```
+--function myFunction() {...6 Lines...}
```

用户可以看到函数中使用了三个以 `v:` 开始的变量, 这些变量由 **Vim** 设置, 分别包含了:

- `v:foldstart`: 被折叠的第一行的行号

- `v:foldend::` 被折叠的最后一行的行号
- `v:foldddashes:` 为每一层折叠包含一个连字符

最后一个变量指出了折叠所在的层次. 假设有这样一段代码:

```
if (x != y) {
    if (y != x) {
        print "x not y"
    }
}
```

最内层 `if` 的 `v:foldddashes` 将会包含 `--` (第二层), 而最外层 `if` 的 `v:foldddashes` 则是 `-` (第一层).

折叠行末尾的连字符是自动加上去的. 如果用户希望使用其他字符, 比如等号, 可以这样做:

```
:set fillchars=fold:=
```

用户可能会感到奇怪为什么需要连字符? 当中的原因其实非常明显. **Vim** 还有一个称为 `foldcolumn` 的折叠设置选项, 这个选项说明了应该使用文本左边的多少列来显示折叠信息. 选项实际上是用这些列画出一个 ASCII 折叠树, 折叠中的连字符就是树叶. 例如:

```
| some text
+- a first level fold
|
|   beginning of open fold
2  indication of fold level
2    - do -
- open fold beginning level 1
+-- a second level fold.
| more text
| more text
```

正如用户所看到的那样, **Vim** 用 ASCII 字符画出了一棵树:

```
|
+-
|
+--
|
+-
+--
```

下面的命令可以设置树的宽度:

```
:set foldcolumn=n
```

命令中的 *n* 是一个 0 到 12 之间的数, 如果折叠的层次比较少, 推荐使用 1 或者 2; 否则的话, 推荐使用 3 到 5.

用户可以对打开或关闭着的所有折叠执行某个命令:



- `:folddoopen cmd`: 对所有未在关闭的折叠中的行^a 执行命令 *cmd*
- `:folddoclose cmd`: 对所有处在关闭的折叠中的行执行命令 *cmd*

^a不仅是处在打开的折叠中的行, 也包括不包含折叠信息的行

4.7.1 提取大纲

当用户使用 **Vim** 编写一个简单的文本文件时, 可能会突然意识到文件已经写得又长又乱了, 这时候就很需要为文件列一个提纲. 这一节介绍如何使用折叠来为文件提取大纲, 提取大纲对于改善文件的结构至关重要.

假设文本文件的内容是:

```
Chapter 1
Section 1 - Vim help
here is some text about the vim help system.
Section 2 - vim scripts
this section contains info about vim scripts.
```

现在, 用户想要折叠文本, 使得只有 **Section** 的头部显示出来. 如果用手动折叠 (`:set foldmethod=manual`) 则比较方便, 方法是选中 **Section** 内的所有行 (包括头部), 然后按下 `zf`, 就可以把这些行折叠起来. 另外, 用户可以认为 **Chapter 1** 是第一层折叠, 每一个 **Section** 是第二层, 关闭 **Section** 所在的折叠后应该变成:

```
-Chapter 1
+Section 1 - Vim help      (2)
+Section 2 - Vim scripts   (4)
```

为了达到上面的效果, 需要做如下设置:



```
:set foldcolumn=1
:set fillchars=fold:\ "there is a space after
the \
:set foldtext=getline(v:foldstart).'      ('.v:
foldstart.')
```

经过这样的处理之后, 文件看起来就像是书籍的目录, 不同之处是这仅仅是一个普通的文本文件而已. 只要新增的行是在之前的行的后面, **Vim** 就仍然会把新增的行当作是折叠的一部分.

如果想要删除折叠, 只需要在可视模式下选中文本, 再按下 `zd`.

如果用户想让文本拥有不同的格式 (比如, 用 `=` 包围 **Section** 的头部), 这当然可以实现 — 只要用户标记了自己的折叠区域, 那么实现起来就不会有什么问题.

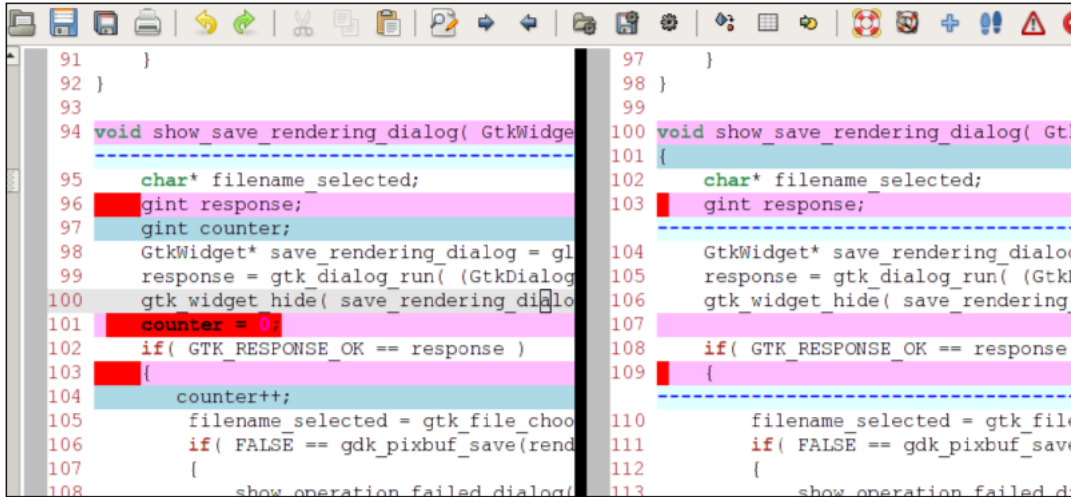
4.7.2 使用 `vimdiff` 比较差异

有时候, 用户可能拥有同一文件的多个版本 — 这些版本之间可能相同, 也可能不同. 在 **Unix** 系统中, 有一个称为 `diff` 的命令, 这个命令很早就开始被人们使用 (最早出现于 1974 年), 但是其他系统很可能没有该命令. 这

个命令的功能是显示两个文件的差异, Vim 也提供了一个用于比较差异的工具 — vimdiff. 这一节介绍如何使用 Vim 比较文件的差异。

112

vimdiff 是 Vim 内置的差异比较工具, 它使用颜色显示文件间的差异 (被比较的两个文件分别处在两个纵向切割或横向切割的窗口中)。下面的图片显示了 vimdiff 的工作界面:



有若干种方式可以用来启动 vimdiff. 在安装了 Vim 的系统中, 一般都有一个叫作 vimdiff 的程序快捷键, 用户可以通过它启动 vimdiff:

```
vimdiff file1 file2
```

上面的命令行等价于:

```
vim -d file1 file2
```

启动 vimdiff 时至少需要提供两个文件进行比较, 至多可同时比较 4 个文件。

如果用户已经启动了 Vim, 并且想要激活 diff 模式, 那么可以这样做:

- `:diffsplit filename`: 水平切割窗口, 并在其中一个窗口打开 *filename*, 所有和 vimdiff 相关的设置会同时应用到这两个窗口中。
- `:vert diffsplit filename`: 垂直切割窗口, 并在其中一个窗口中显示 *filename*, 所有和 vimdiff 相关的设置会同时应用到这两个窗口中。
- `:diffthis`: 使得当前窗口成为差异窗口的一部分。当用户想要再次对某个文件进行差异比较时, 可能会用到该命令。

113

一个常见的例子是比较当前文件, 与上一次 Vim 所保存的备份之间的差异 (备份文件的文件名以波浪号 ~ 结尾)。假设用户正在编辑的文件是 `main.c`, 则可以在任意时刻保存文件, 然后执行:

```
:vert diffsplit main.c~
```

执行命令后, 窗口被垂直分割成两个, 文件间的差异通过颜色标记出来 (具体的颜色取决于当前使用着的色彩主题)

还有一种情况是, 对文件作出修改的并不是用户本人, 而是另一个开发人员向用户发了一个文件补丁, 对于这种情形, Vim 也提供了一个 diff 视图, 用于查看补丁被应用到文件后所带来的变化。方法是打开补丁所对应的文件, 然后执行 `vert diffpatch patchfile`, 或者省略 `vert`, 直接执行 `:diffpatch patchfile`。

命令中的 *patchfile* 是其他开发人员发送给用户的补丁. 执行命令后, Vim 会在另一个窗口中打开文件, 把补丁打到该文件上, 然后比较打补丁前, 与打补丁后文件的差异.

4.7.3 在 vimdiff 中导航

和普通的 Vim 窗口相比, 在 vimdiff 窗口中导航会稍有不同.

例如, 滚动其中一个窗口时, 会同时滚动另一个窗口. 实际上, 窗口之间的每一行, 其相对位置不变, 这是通过选项 *scrollbind* 来实现的:

- 开启同时滚动的命令是:

```
:set scrollbind
```

- 关闭同时滚动的命令是:

```
:set noscrollbind
```

当在 diff 窗口中编辑文件时, Vim 会自动根据文件内容的变化, 来相应地更新窗口中的颜色; 如果没有更新, 则可以执行:

```
:diffupdate
```

如果光标正处在某一 diff 窗口中, 而用户希望在修改之间快速地跳转, 这就需要用到下面的命令:

- [c: 跳转到前一个修改的开始
-]c: 跳转到下一个修改的开始

通过这种方法, 用户可以在文件的相关区域间方便地跳转, 并且可以很容易看出对文件作了哪些修改.

如果光标正处在 vimdiff 窗口的某一差异上, 而且用户知道该修改对文件的另一版本也是需要的. 虽然可以把修改复制下来, 并插入到另一个文件的适当位置, 但是 Vim 提供了比这更方便的做法. 下面这个命令可以把修改应用到同一文件的另一个版本上:

```
:diffput
```

执行该命令时, 光标必须处在待应用的修改上. 另一方面, 如果光标是处在不含有修改的文件上, 那么用户可以先跳转到含有修改的文件上, 然后把修改应用到另一个文件, 或者直接使用下面的命令:

```
:diffget
```

除了这两个命令, 还可以在普通模式下使用另外两个替代命令 *do* 与 *dp*.



更多的内容请参考 `:help vimdiff`.

4.7.4 使用 diff 跟踪变化

前面一节已经学习到如何使用 `vimdiff` 来比较同一文件的不同版本之间的差异, 但是, 如果用户仅仅是想知道在保存之前, 对当前活动缓冲区作了哪些修改, 那又应该怎么办呢? 这一节介绍如何查看硬盘上的文件与缓冲区中的文件之间的差异, 也就是自从上一次保存以来, 文件所发生的变化. 用户所要做的首先是把下面的代码添加到文件 `vimrc`:

```
function! DiffWithFileFromDisk()
    let filename=expand('%')
    let diffname = filename.'.fileFromBuffer'
    exec 'saveas! ' .diffname
    diffthis
    vsplit
    exec 'edit ' .filename
    diffthis
endfunction
```

函数为当前缓冲区中的文件保存一份临时副本 (包含了最新的修改), 然后对临时文件与硬盘上的文件进行差异比较.

为了调用这个函数, 需要执行:

```
:call DiffWithFileFromDisk()
```

也可以为调用命令绑定一个快捷键:

```
:nmap <F7> :call DiffWithFileFromDisk(<cr>
```

上面的命令把功能键 **F7** 绑定到函数调用, 使用时只需要切换到普通模式, 再按下 **F7**, 就可以看到用 `diff` 模式标记的修改.

现在, 用户就可以在保存文件之前, 查看所有未保存的修改, 检查是否每个修改都是正确的.

4.8 打开任意位置的文件

系统管理员与网站开发人员经常会被同一个问题所困扰, 如果解决得不好的话, 会给人带来很大的麻烦, 这个问题就是他们所浏览的文件大部分都存放在远程服务器上.

系统管理员对此问题的解决办法通常是远程登录到服务器上 (比如用 `ssh` (`Secure Shell`), 然后再在服务器上直接修改配置文件.

网站开发人员的解决办法通常是先把文件下载到本地, 编辑好后再上传到服务器上, 下载与上传的工具可以用 `FTP` 客户端, 或 `Webdav`.

如果不使用上面提到的方法, 那又应该怎么办呢? 为什么他们就不能在本地直接编辑远程服务器上的文件? `Vim` 可以在不需要额外扩展的前提下完成这件事. `Vim` 含有一个称为 `netrw` (`Net Read/Write`) 的系统, 它可以用来编辑远程服务器上的文件, 通过例子介绍如何使用它.

假设有一个网站开发人员, 叫作 `John`, 他把他的主页放在远程服务器 `remote.server.com` 上. 现在他想要编辑文件 `index.html`, 这个文件存放在服务器家目录下的 `public_html/`. 使用 `Vim` 打开此文件的命令是:

```
vim ftp://john@remote.server.com/public_html/index.html
```

Vim 根据参数识别出它需要使用 FTP 协议连接到服务器 `remote.server.com`, 所使用的用户名是 `john`, 如果还需要输入密码, Vim 就会在连接时给出提示信息. Vim 从服务器上下载一份文件的副本到本地, 用户所编辑的正是该副本, 所不同的是每当用户保存文件时, Vim 都会同时保存一份到服务器上.

如果 John 已经打开了 Vim, 则可以使用下面两个命令中的一个打开远程服务器上的文件:

```
:Nread ftp://john@remote.server.com/public_html/index.html
:Nread remote.server.com john PASSWORD public_html/index.html
```

命令中的 PASSWORD 是登录时所使用的密码. 除了可以读取远程服务器上的文件, 还可以把本地文件写到远程服务器上, 甚至可以在一台服务器上打开文件, 并把它保存到另一个服务器上. 把文件保存到 FTP 服务器的命令是:

```
:Nwrite ftp://user@server/path/to/filename
:Nwrite server user password path/to/filename
```



针对不同的协议, :Nread 与 :Nwrite 所使用的参数也有所不同, 可以通过 :Nread ? 与 :Nwrite ? 查看具体的规则.

除了 FTP, Vim 还支持以下协议:

- SCP
- SFTP
- RCP
- HTTP (只读)
- DAV
- rsync (只读)
- fetch (只读)

为了在 Vim 的命令中使用这些协议, 只需要把前面例子中的 `ftp` 替换成对应的协议名 (小写形式) 即可.

然而, 有一点需要注意. 为了使用这些协议, Vim 依赖于外部的命令程序. 在 Linux 系统中, 这些程序默认都是可用的, 但是对于 Microsoft Windows 来说, 只有 FTP 是可用的. 为了查看 Vim 所使用的外部程序, 以及如何修改它们, 执行:

```
:help netrw-externapp
```

除了读取与编辑远程文件, Vim 还可以列出远程目录中的所有文件, 这做便于用户选择正确的文件进行编辑. 为了列出远程目录中的文件, 只需要把 :Nread 的参数由文件改为目录即可, 例如:

```
:Nread scp://user@server/some/directory/
```

执行命令后, 用户可以选择任意一个文件进行编辑, 就好像它们就在本地目录.



如果是 Linux 系统, 则用户可以把登录远程服务器所用的用户名与密码保存到本地家目录的 `.netrc` 文件中, 更多的信息参考 :help netrw-netrc.

4.8.1 更快的远程文件编辑

用户既然已经知道了如何直接编辑远程服务器上的文件,那么他就很有可能同时打开多个远程文件.这时用户就会遇到一个很恼人的情况:每当移动到另一个缓冲区时(用命令 `:bufferprev` 或 `:buffernext`),都需要重新登录.

默认情况下,每次显示缓冲区的文件内容时, **Vim** 都会尝试重新加载文件.也就是说,如果打开的是一个远程文件,那么 **Vim** 就会尝试重新登录,以检查文件是否需要重新加载.

但是这真的有必要吗?如果远程文件除了用户之外,没有其他人正在编辑,那么在切换缓冲区时就不需要重新加载文件.

每一个缓冲区都有一套选项,指明了在不同的情况下如何处理缓冲区.其中之一就是 `bufhidden`,它表示当缓冲区被隐藏(在窗口中不可见)时 **Vim** 该做什么操作.默认情况下该选项的值为空,如果把它设置成 `hide`,那就是告诉 **Vim** 当在窗口中看不到缓冲区时,只需把它隐藏即可;当再次把它显示在窗口中时,只需把它显示出来即可.为了不让 **Vim** 在切换缓冲区时重新从远程加载文件,在 `vimrc` 中添加:

```
set bufhidden=hide
```

118

4.9 小结

这一章介绍了如何在日常工作中更高效地使用 **Vim**.介绍了很多方法,也提出了一些优化措施.

首先是介绍了如何利用模版来降低文本输入的工作量.最开始是使用缩写来插入模版,然后,是根据不同的文件类型来制作并插入对应的模版文本.最后,讨论如何使用 `snipMate` 脚本来为任意的文件格式制作高级 `snippet`.

接着是自动补全.我们介绍了各种不同的补全方法,还说明了如何把众多补全方法都绑定到 `Tab` 键上.

通过记录一系列命令,用户就可以多次重复执行同一段命令序列.说明了如何使用宏录制,把日志文件的内容转换成 `HTML` 格式.

再接下来是 **Vim** 的会话.介绍了如何保存窗口的状态,以及如何把会话用作项目管理程序.

通过寄存器,用户可以使用多达 9 个不同的寄存器/剪贴板.

通过折叠文本,用户能够得到更好的文件概览,因为无关紧要的部分都被隐藏在了折叠中.还可以利用折叠来制作文件的提纲.

文件被修改过了,但是具体改了哪些地方呢?我们介绍了如何使用 **Vim** 内置的 `diff` 功能.通过它,用户可以获取文件修改的概览,或者是在维持良好的 `diff` 概览的前提下,撤消或添加分支.

编辑本地文件是一回事,编辑远程机器上的文件是另一回事. **Vim** 可以做到直接编辑或操作远程文件,而且和编辑本地文件相比,用户并不会感到有什么不同.

在下一章,我们将会学习到如何使用 **Vim** 的格式化选项来格式化普通文本或代码.

119

第五章 格式化进阶

121

很多时候,最简单的工作就是把文本或代码修改成更易阅读的形式.本章将会介绍一些简单的方法,用于对文本进行格式化 — 无论是普通文本还是代码.

这一章主要分为三个部分:

- 文本格式化
- 代码格式化
- 使用外部格式化工具

学完这一章后,用户应该可以清楚地知道当要对文本进行格式化时,什么情况下应该用 Vim,什么情况下不应该用 Vim.

5.1 格式化文本

在编辑普通文本时,虽然大多数人更喜欢使用图形化字处理软件,比如 Microsoft Word 或 OpenOffice,但是许多纯文本编辑器,比如 Vim,也可以把事情做得很好.在下面的一节里,将会介绍如何利用 Vim 强大的功能来格式化普通文本.

122

5.1.1 文本分段

这一节所介绍的知识可能是整本书中最简单的,但是对于格式化普通文本来说,却可能是最有用的.假设用户正在编写一段文本,并且在编写的过程中丝毫没有注意到行的变化与文本的格式.最终,用户可能会写出一行非常长的文本,此时他才注意到应该对文本重新进行格式化,摆在他面前的有两个选择:

- 遍历文本,并在适当的地方断行
- 用某个命令对整段文本进行格式化

很显然,后者是最好的选择,并且格式化后的结果也可以和其他结果保持一致.所使用的命令是:

```
gqap
```

上面的命令其时是两个命令的组合:

- gq: 把紧接在该命令后的动作所覆盖到的文本进行格式化
- ap: 选择一个段落 (“a paragraph”)

换句话说, 由 `gq` 与 `ap` 组合而成的命令告诉 Vim 去遍历当前段落并格式化. 由两个空白行包围起来的部分定义为一个段落, 因此, 为了开始一个新的段落, 只需要添加一个空行即可.

Vim 所做的格式化实际上是让行作出更漂亮的断行, 使得每一行的长度都不会超过某个特定的大小 (Vim 自动地在适当的两个单词间作出断行).

格式化后文本的宽度由选项 `textwidth` 定义, 如果用户希望每行最大的宽度不超过 80 个字符, 那就在 `vimrc` 中添加:

```
:set textwidth=80
```

如果 `textwidth` 的值被设置为 0, 那么 Vim 就把它设置成窗口的宽度 — 但是永远不会多于 `textwidth` 所设置的字符个数.^①

123



通过设置选项 `formatoptions` 可以控制 Vim 如何格式化段落. 更多的信息可以参考 `:help 'formatoptions'` 与 `:help 'fo-table'`.

`gq` 可以和任意的移动命令配合使用, 并且在格式化之后, 光标将会停留在移动命令结束的地方 (典型的情况是停留在当前特定区域的最后一行). 如果用户希望在格式化后, 光标仍旧处于格式化开始前的位置, 那就把 `gq` 改成 `gw`. 如果用户的光标原来是在段落中第一行的开始, 此时执行 `gwap`, 命令结束后, 光标仍然会停留在段落中第一行的开始.

用户可以在命令的前面加上数字, 使得它重复执行, 例如, `5gqap` 将会对当前与下面的四个段落进行格式化. 如果要对文件内的所有段落进行格式化, 就执行 `1gqG`.

前面的介绍的格式化命令不仅对普通文本有效, 对其他任意类型的内容同样有效, 而且用户可以决定使用哪种格式化函数.

用户可以为指定的文件类型设置任意的格式化函数, 方法是设置 Vim 的选项 `formatexpr`. 例如, 如果用户想要对 C 源代码进行格式化, 只需要在 `vimrc` 中添加:

```
:set formatexpr=c#Formatter()
```

这行命令告诉 Vim, 在打开一个 C 源代码文件时, 使用函数 `Formatter()`, 这个函数定义在 Vim 为 C 文件自动加载的文件中.



自动加载的文件可以在 `VIMHOME` 的子目录 `autoload` 中找到. 文件根据所服务的文件类型来命名, 以后缀 `.vim` 结束. 例如, 为 C 文件自动加载的文件是 `VIMHOME/autoload/c.vim`.

一个格式化函数含有三个变量, 利用这三个变量可以找到待格式化的文本.

- `v:num`: 待格式化的第一行的行号
- `v:count`: 待格式化的行的数量
- `v:char`: 这个变量包含了将被插入的字符, 可以为空

格式化函数的一个简单示例是:

124

^①原文是 If the option is set to 0, the Vim sets it to the width of the window — but never more than the number of characters defined in the `textwidth` setting.

```
function! MyFormatter()
    let first = v:num
    let last = v:num + v:count
    while(first<=last)
        call setline(first, '> '. getline(first))
        let first = first+1
    endwhile
endfunction
```

这个格式化文本接收所有的行, 在每一行的开始添加 >, 就像电子邮件中的引用。

上面展示的格式化函数非常简单, 如果需要更高级一点的, 那么函数的复杂度就会快速增加, 因此公开可用的格式化函数非常有限 (这些函数是为了某些特定的目标而开发的)。

5.1.2 对齐文本

在大多数字处理程序中, 最基本的格式化选项之一是左对齐, 右对齐, 或居中。其中一些程序甚至可以让文本两端对齐, 这样做可以让每一行的结束尽可能地向边缘靠近。

虽然上面提到的格式化类型对字处理程序来说非常常见, 但对于普通文本编辑器来说就很少见了, Vim 就位列其中。

Vim 支持三种类型的对齐 — 左对齐, 右对齐, 居中。在介绍它们如何工作之前, 先简单解释一下支持这种对齐类型的文本编辑器为什么很少。

对于常见的文本编辑器来说, 它们不含有隐藏信息, 也就是说, 用户所能看到的, 就是用户所能拥有的 — 没有页面宽度, 没有对齐, 什么都没有。

另一方面, 在字处理程序中, 文档隐藏了相当丰富的信息, 这些信息告诉编辑器如何根据用户的需要, 对文本进行格式化。

因为上面所说的情况对 Vim 并不适用, 所以用户得向 Vim 提供一些信息, 来帮助 Vim 进行格式化。例如, 为了让编辑器知道对齐时所需的边缘, 用户需要设置文本的宽度。

说完了这些, 先来看一下居中排列的命令:

```
: [range] center WIDTH
```

命令中的 range 是用户希望居中的行的范围, WIDTH 是每行最多的字符数。典型的用法是在可视模式下选中待居中的文本 (按住 *Shift+v*, 然后移动光标), 然后再输入命令。按下 : 后, 用户将会发现 Vim 自动地把选中的文本的范围记成 '<', '>', 这表示从选中文本的第一行到最后一行。

接下来, 用户需要输入 center 与文本的宽度。如果选项 textwidth 已经设置妥当, 那就不需要输入 WIDTH。

如果选项 textwidth 的值为 0, 用户又没有在命令中设置 WIDTH, 那么 Vim 就默认使用 80 个字符的宽度。在字处理程序中, 用户很容易就可以看出文本是否是居中的, Vim 对文本进行居中时, 只是在前面加上适当数量的空格, 这样做意味着无论是在什么时候修改了文本, 用户都要重新对文本进行居中。

下面的命令用于左对齐:

```
: [range] left INDENT
```

同样, 执行这个命令时需要提供行的范围, 如果需要的话, 再提供缩进的宽度。这个命令可以用来精确地设置文本的左边边缘。

最后是右对齐命令. 同样的, Vim 可能并不知道一行的宽度, 因此执行命令时还要提供宽度信息. 命令的形式是:

```
: [range] right WIDTH
```

范围内的行用空格缩进, 使得每一行的结束都是对齐的, 宽度由用户指定. 和居中一样, 无论何时修改了文本, 都要重新对文本进行右对齐.

5.1.3 标记标题

使用普通的文本编辑器编写文档时, 用户可能需要创建自定义的格式或标记, 使得文本更容易阅读.

为了提高可读性, 一个常见的操作是为那些用作标题的字符串作标记, 这些标题可以是文本的章节.

如果是字处理程序, 只需要把字体设置得更大更粗就可以了, 但是这样的字体设置对 Vim 却是不可能实现的, 因为 Vim 字体的大小是固定的. 因此, Vim 用户必须通过其他的方法来标记标题.

笔者个人的选择是在标题的下面添加一行, 来表示这是一个标题行.

一个简单的例子是:

```
My Headline
=====

This is the text on the document. It could contain one
or more lines of text.
```

不同级别的标题可以用不同类型的标记来表示:

```
Level1
=====
Level2
-----
-Level3-
```

为了更方便地添加下划线, 在 Vim 中可以用宏来实现, 使用这种方法不用担心下划线添加得太多或太少. 前面两个级别的标题宏可以实现成:

```
yypVr=o
```

宏的各个部分的具体涵义是:

- yy: 复制当前行
- p: 粘贴
- v: 选中一整行
- r: 用后面的字符 (在这里是 =) 替换掉选中的字符
- o: 在光标的下面添加一个空行, 把光标移动到这行的开始, 并切换到插入模式

这个宏的基本功能是获取当前行 (标题行), 并复制它. 然后把复制得到的行的所有字符替换成某个字符 (在这个例子中是 - 或 =), 最后, 插入一个新行, 并切换到插入模式.

对第三级别的标题来说, 我们必须采取其他办法, 所做的操作无非是在一行的开始和结束添加一个连字符, 可以用一条替换命令来完成:

```
:s/\.(.*\)/-\1-/
```

把命令拆成三部分来说明:

- `:s///`: 替换命令
- `\.(.*\)`: 正则表达式, 表达式把当前行的所有字符作为输入, 并把它们作为搜索模式
- `-\1-`: 替换模式. 替换模式告诉 Vim 在第一个被匹配的子模式前添加一个连字符, 在子模式的后面再加上一个连字符

记住, 这些宏可能会写得很复杂, 但我们可以很轻易地为它们绑定一个快捷键, 例如:

```
:map h1 yypVr=0
:map h2 yypVr-0
:map h3 :s/\.(.*\)/-\1-/
```

现在, 用户可以在普通模式下, 通过按 `h1`, `h2`, `h3` 来添适当的标题行. 如果用户不想在标题行的下面添加一个空行并切换到插入模式, 就把上面的映射命令中的 `o` 删掉.

5.1.4 创建列表

项目列表与编号列表是文档中的常见结构. 本节将会介绍在 Vim 中如何方便地创建这些列表.

首先来看一个函数, 这个函数接收某个范围内被选中的行, 然后把它们转化成项目列表. 在这个列子里, 一个简单的项目列表是:

```
* first item
* second item
* third item
```

为每一行的开始添加星号的函数可以是:

```
function! BulletList()
    let lineneno = line(".")
    call setline(lineneno, " * " . getline(lineneno))
endfunction
```

认真看一下函数的实现, 就会发现它所做的不过是获取当前行, 然后再用它的一份拷贝替换掉自身, 在前面添加一个空格, 一个星号, 然后是一个制表符.

很显然, 函数只对一行有效, 然而, 如果用户选中了一个范围内的所有行, 那么 Vim 就会为每一个选中的行调用一次函数, 从第一行, 一直到最后一行. 如果是为每一行添加相同的内容, 那么这种方法就很方便.

然而, 上面介绍的方法并不适合编号列表, 因为用户必须记住下一个号码从多少开始.

所以, 现在来看下面的函数, 它把某个范围内选中的行转换成编号列表 — 列表的每一项都占据一行:

```
function! NumberList() range
    " set line numbers in front of lines
    let beginning=line("'<")
    let ending= line("'>")
```

```

let difsize = ending-beginning +1
let pre = ' '
while (beginning <= ending)
    if match(difsize, '^9*$') == 0
        let pre = pre . ' '
    endif
    call setline(ending, pre . difsize . "\t" . getline(ending))
    let ending=ending-1
    let difsize=difsize-1
endwhile
endfunction

```

这个函数稍微有点复杂, 但和它所解决的问题比起来, 还是比较简单的 — 在每一行的开始添加一个数字. 除此之外, 函数还做了一个额外的工作 — 数字右对齐:

```

1 item1
2 item2
...
10 item10
11 item11
...
100 item100
...

```

为了做到这样的对齐, 需要考虑两个问题:

- 必须知道列表最大的编号
- 必须一次处理所有的行

为了解决第一个问题, 必须查看范围内第一行与最后一行的行号, 两个行号之间的差就是行数, 于是就得到了列表编号的最大值. 由于考虑了第二个问题, 所以这是唯一可能的解决办法, 如果不这样做的话, 那么函数每次就只能看到当前行.

解决第二个问题的办法是在函数名的后面添加关键字 `range`, 这个关键字告诉 Vim, 函数是对一个范围内的行作操作, 而不仅仅是一行.

函数从范围内的最后一行处理到第一行, 无论何时碰到一个只含有 9 的数字 (比如 99, 9999), 都说明编号中的字符少了一个 (例如, 从第 1000 行到第 999 行), 为了弥补缺少的一个字符, 函数为缩进新增一个空格. 通过这种方法, 可以一直保证数字的右对齐, 无论范围内有多少行.



在 <http://www.vimoutliner.org>, 用户可以找到利用标题格式化, 列表格式化等技术来为文档列提纲的脚本. 如果用户需要在 Vim 中为文档列提纲, 最好试一下这个脚本.

5.2 格式化代码

在对代码进行格式化时经常需要考虑很多因素. 每一种语言都有自己的语法规则, 还有些语言对格式非常依赖. 在有些情况下, 程序员需要按照公司给出的规定来格式化代码.

那么, Vim 如何知道用户想把代码格式化成什么样子? 简单来说 Vim 并不需要知道. 但是 Vim 有办法让用户完全按照自己的需要来格式化代码.

130

虽然具体的格式化细节不尽相同, 但是它们都遵循一些基本的规则, 也就是说, 用户只需要关心不同的地方即可. 在大部分情况下, 格式化规则的修改可以通过一系列的 Vim 选项设置, 在这些选项当中, 比较重要的有以下几个:

- `formatoptions`: 这个选项负责特定格式的设置 (见 `:help 'fo'`)
- `comments`: 什么是注释, 以及如何对它们进行格式化 (见 `:help 'com'`)
- `(no)expandtab`: 用空格代替制表符 (见 `:help 'expandtab'`)
- `softtabstop`: 一个制表符可以用多少个空格来代替 (见 `:help 'sts'`)
- `tabstop`: 一个制表符的宽度 (见 `:help 'ts'`)

通过这些选项, 用户几乎可以设置与缩进相关的方方面面. 但是光有这些还不足够, 用户仍然需要告诉 Vim 是否需要自动缩进, 还是由用户手动完成缩进. 如果用户希望由 Vim 来完成缩进, 可以通过 4 种方法来完成, 下面的小节将会介绍与代码缩进相关的一系列选项.

5.2.1 Autoindent

Autoindent 是缩进代码最简单的方式. 它的功能仅仅是与上一行保持相同的缩进层次. 所以说, 如果当前行缩进了 4 个空格, 按下 *Enter* 后插入的空行也会缩进 4 个空格. 至于什么时候修改缩进的层次则完成取决于用户. 这种缩进方式适用于若干行需要保持相同缩进层次的语言. 打开 **autoindent** 的命令是 `:set autoindent` 或 `:set ai`.

5.2.2 Smartindent

Smartindent 比 **autoindent** 稍微智能一些. 它仍然可以让上一行的缩进层次保持到下一行, 但用户无需手动修改缩进层次. **Smartindent** 可以识别出 C 语言的大部分结构, 并根据它们来决定何时增加/减少缩进层次. 由于许多编程语言都或多或少地都继承了 C 语言的语法, 所以 **smartindent** 也可以应用到其他语言. 打开 **smartindent** 可以用下面两个命令中的任意一个:

- `:set smartindent`
- `:set si`

131

5.2.3 cindent

Cindent 经常被称为 **clever indent**(聪明的缩进) 或 **configurable indent**(可配置的缩进), 这是因为与前面介绍的两种缩进相比, 它的可配置性更强. 有三种设置选项:

- **cinkeys**: 这个选项包含了一个列表, 列表中的各项之间用逗号分开, Vim 可以根据列表中的项来改变缩进层次. 一个例子是: `:set cinkeys="0{,0},0#,:"`, 意思是说无论何时碰到一个以 {, 或 }, 或 # 作为开始的行, 或者是以 : 作为结束的行 (很多语言的 switch 结构都用到了 :), Vim 都要再缩进一层. cinkeys 的默认值是 `"0{, 0}, 0), :, 0#, !^F, o, O, e"`, 更多的信息见 `:help cinkeys`.
- **cinoptions**: 这个选项包含了所有的, 专门用于 **cindent** 的选项. 它是一个各项之间由逗号分开的列表, 通过这个列表可以设置大量的选项. 一个例子是 `cinoptions=">2,{3,}3"`, 意思是说在正常的缩进之上, 再额外添加两个空格, 另外, 在 { 与 } 的左边添加三个空格, 以便与前一行作比较. 因此, 如果目前的缩进是 4 个空格, 那么刚才所说的设置会使代码变成:

```
if_(a_==_b)
    __{
        _print_"hello";
    _}
```

cinoptions 的默认值是 `">s,e0,n0,f0,{0,}0,^0,:s,=s,l0,b0,gs,hs,ps,ts,is,+s,c3,C0,/0,(2s,us,U0,w0,W0,m0,j0,)20,*30"`, 更多的信息见 `:help 'cinoptions'`.

- **cinwords**: 这个选项包含的关键词会让 Vim 在下一行增加缩进. 一个例子是 `:set cinwords="if, else, do, while, for, switch"`, 这同时也是它的默认值. 更多的信息见 `:help 'cinwords'`.

132

5.2.4 Indentexpr

Indentexpr 是最灵活的缩进选项, 但同时也是最复杂的. 使用 **indentexpr** 时, 它会对一个表达式求值, 然后计算出一行的缩进. 因此, 用户写出的表达式必须能被 Vim 求值. 打开这个选项的方法是为它设置一个表达式, 比如:

```
:set indentexpr=MyIndenter()
```

命令中的 `MyIndenter()` 是一个函数, 负责计算行的缩进.

函数的一个简单例子是模仿选项 **autoindent** 的功能:

```
function! MyIndenter()
    " Find previous line and get it's indentation
    let prev_linen = s:prevnonblank( v:lnum)
    let ind = indent( prev_linen )
    return ind
endfunction
```

即使是想把函数的功能增强一点, 函数的复杂度也会迅速增长. Vim 自带了多种编程语言的缩进表达式, 如果用户想要自己编写缩进表达式, 可以以它们为基础, 再加以修改. 用户可以在 `VIMHOME` 的子目录 `indent` 中找到实现代码.

更多的信息见 `:help 'indentexpr'` 与 `:help 'indent-expression'`.

5.2.5 代码块快速格式化

缩进选项设置完毕后, 用户可能想根据新设置的选项更新一下代码的格式. 为了更新代码, 只需要告诉 Vim 重新缩进第一行到最后一行, 具体的命令是:


```
1G=G
```

如果把命令拆开来看,各个部分的意思是:

- 1G: 跳到文件的第一行 (也可以使用 gg)
- =: 根据格式化的配置对文本加以缩进
- G: 跳到文件的最后一行 (缩进的结束位置)

如果把命令映射到按键上,使用起来就更加方便:

```
:nmap <F11> 1G=G
:imap <F11> <ESC>1G=Ga
```

后一个命令尾巴的 a 是为了在格式化完成后,重新回到插入模式,因为在执行命令前就处在插入模式下. 现在,只需按下 *F11*,就可以对整个缓冲区的文本重新缩进.



注意,如果待缩进的代码含有编程错误,比如在 C 代码中漏写了一个分号,那么代码就不会被正确地缩进. 利用这个性质可以在一定程度上检查编程错误.

有时候,用户可能仅仅是想格式化某一小块代码,对于这种情况,用户可以使用代码块自然形成的范围,或者是在可视模式下选择一段代码,然后再格式化.

后一种方法比较简单. 先是切换到可视模式,比如按下 *Shift+v*,然后再按 = 来重新缩进被选中的代码行.

另一方面,如果要使用代码块,则有多种不同的方法来实现. 在选择代码块时, Vim 提供了很多的方法. 为了与一个缩进代码的命令作组合,我们必须介绍这些不同类型的代码块,以及选择它们的命令:

- i{: 内部块 (Inner block), 指的是 { 与 } 之间的所有内容 (不包括 { 与 }). 还可以用 i} 与 iB 来选择.
- a{: 一个块 (A block), 指的是 { 与 } 之间的所有内容 (包括 { 与 }). 还可以用 a} 与 aB 来选择.
- i(: 括号内 (Inner parenthesis), 指的是 (与) 之间的所有内容 (不包括 (与)), 还可以用 i) 与 ib 来选择.
- a(: 一对括号 (A parentheses), 指的是 (与) 之间的所有内容 (包括 (与)), 还可以用 a) 与 ab 来选择.
- i<: 内部 <> 块 (Inner <> block), 指的是 < 与 > 之间的所有内容 (不包括 < 与 >), 还可以用 i> 来选择.
- a<: 一个 <> 块 (A <> block), 指的是 < 与 > 之间的所有内容 (包括 < 与 >), 还可以用 a> 来选择.
- i[: 内部 [] 块 (Inner [] block), 指的是 [与] 之间的所有内容 (不包括 [与]), 还可以用 i] 来选择.
- a[: 一个 [] 块 (A [] block), 指的是 [与] 之间的所有内容 (包括 [与]), 还可以用 a] 来选择.

现在,我们已经知道了 Vim 是如何看待一个代码块的,接下来,只需要告诉它如何操作代码块,在这个例子中,是希望 Vim 重新缩进代码块. 前面已经说到, = 可以用来重新缩进代码,所以重新缩进代码块的一个示例是:

```
=i{
```

用上面的命令重新缩进下面的代码 (| 表示光标的当前位置):

```
if( a == b )
{
    print |"a equals b";
}
```

执行命令后, 代码变成 (默认使用 C 语言的缩进格式):

```
if( a == b )
{
    print |"a equals b";
}
```

如果改用 `a{` 来选择代码块, 代码就会变成:

```
if( a == b )
{
    print "a equals b";
}
```

在最后一段代码中, 命令 `=a{` 同时纠正了花括号与打印语句的缩进.

有时候, 用户可能会遇到一个带有多级代码块的代码块, 并且用户希望重新缩进当前代码块及包围它的代码块. 不用担心, **Vim** 提供了一个很方便的做法. 比如, 如果用户想要重新缩进当前代码块及包围它的代码块, 只需要把光标放在相对内层的代码块上, 并执行:

```
=2i{
```

这个命令告诉 **Vim** 重新缩进内层代码块中的两层, 从当前“活跃”的块到该块的外层. 可以把命令中的 `2` 替换成用户想要重新缩进的层数. 当然, 还可以把 `i` 替换成其他的代码块选择命令, 从而精确地选择待缩进的代码块.

上面介绍的这些, 就是缩进代码所要掌握的全部内容.

135

5.2.6 自动格式化粘贴的代码

经常有程序员告诉我, 他们会经常复用已有的代码, 这意味着他们经常需要复制与粘贴代码.

许多用户在把代码粘贴到 **Vim** 窗口中时, 都会碰到过“阶梯效应”——当插入代码时, **Vim** 会尝试对代码进行缩进, 造成的结果是越往后, 行的缩进层次越深, 类似于阶梯:

```
code line 1
  code line 2
    code line 3
      code line 4
        ...
```

解决这个问题的办法通常是把 **Vim** 设置成粘贴 (`paste`) 模式, 用到的命令是:

```
:set paste
```

代码粘贴完毕后, 用下面的命令把 **Vim** 设置回正常的插入模式:

```
:set nopaste
```

那么,除此之外,是否还有其他的方法呢? Vim 是否可以根据文件中已有的其他代码来自动格式化被粘贴的代码? 其实很简单.

```
p=`]
```

上面的命令只是把普通的粘贴命令 (p) 和另一命令组合,后者会自动缩进前面插入的行 (=`]). 这个命令基于下面的事实: 当用户使用命令 p 来粘贴文本时,光标将会停留在被粘贴文本的第一个字符上,组合上命令`]之后,光标就会移动到最近一次插入的文本行的最后一个字符上,并且可以从被粘贴文本的第一行移动到最后一行.

所以,用户所要做的就是把这个命令绑定到一个快捷键上,并用该快捷键来粘贴文本.

136

5.3 使用外部格式化工具

即使是经验丰富的 Vim 用户也会经常说 Vim 可以做任何事情,虽然事实上并不是这样,但其实已经很接近了——对于 Vim 不能做的事,最好借助外部工具来完成.

下面的小节介绍最经常使用的格式化工具,以及如何使用它们.

5.3.1 Indent

程序 Indent 可能是最常使用的 Vim 外部工具. 从 80 年代开始,它就被安装到了各种不同的 Unix 平台中,后来还被移植到了 Microsoft Windows.

顾名思义,这个程序的功能就是缩进代码——特别是那些类 C 代码. 用户可会感到奇怪——既然 Vim 本来就可以缩进代码,而且做得也很不错,那干嘛还要用外部工具呢? 这个问题非常好,答案是 Vim 确实在这方面做得很好,但是 Indent 做得更好,并且更容易在多个编辑器之间对缩进进行标准化.

由于 indent 专门用于缩进代码,所以它的效率和 Vim 相比会高出很多,对于后者来说,缩进代码只是一项特性而已,而不是唯一的特性. Indent 对代码的理解力很高,可以根据代码来缩进——即使含有语法错误.

那么,在 Vim 中应该如何使用 Indent? 前面的几节已经介绍了 Vim 关于缩进的几个选项,然而,现在只需要一个选项即可:

```
:set equalprg=program
```

这个选项的功能是当使用命令 = 缩进代码时, Vim 应该使用哪种外部工具. 对于 Indent, 只要把 *program* 替换成程序 Indent 的路径即可. 设置之后,无论何时使用缩进命令 (比如 1G=G), Vim 就会把待缩进的文本输送给程序 Indent. 甚至可以在选项中加上命令行选项.

Indent 有着非常丰富的命令行选项,另外还可以通过它的配置文件来修改 Indent 的行为.

137



可以从以下网址下载到最新版的 Indent: <http://www.gnu.org/software/indent/>.

5.3.2 Berkeley Par

90 年代早期, Adam M. Costello 开始开发一个简单的命令行工具,这个工具的唯一功能是按照用户的要求,重新格式化文本中的段落. 工具的名字是 Par, 一两年后,该工具拥有了非常丰富的功能,几乎可以重新格式化任意类型的段落.

正因为如此,Par 成了 Vim 的理想工具,现在介绍一些使用示例.

假如,用户想重新格式化段落,使得每一行最大长度不超过 78 个字符,那就在 Vim 中执行:

```
:set formatprg=par\ -w78
```

选项 `formatprg` 告诉 Vim, 当使用命令 `gg` 时, 应该调用哪个外部工具来格式化文本. 值得注意的是, 程序和它的参数之间的空格用一个反斜杆转义, 这做就可以让 Vim 把程序与它的参数看作一个整体.



注意, 如果 `formatexpr` 为空, 则只会使用 `formatprg`, 否则的话, 就使用 `formatexpr`.

前面曾经提到过, Vim 无法对文本进行两端对齐, 幸运的是, Par 可以帮助 Vim 完成这件工作. 只需要在上面的命令行参数中加上 `j` (意思是 `justify`), 就可完成两端对齐:

```
:set formatprg=par\ -w78j
```

Par 不仅可以用在普通文本中, 对于代码的某些部分也可以使用, 比如注释.

假设用户拥有下面一段注释:

```
/* *****
/* This function helps you modify a string and remove all */
/* unnecessary characters . */
/* Don't use this on widechar strings or strings shorter than 10 */
/* characters */
/* ***** */
```

用户可以选中这些注释, 然后执行:

```
!par 60r
```

(Vim 会在 ! 的前面自动加上 '`<`', '`>`')

执行后, 注释变成:

```
/* *****
/* This function helps you modify a string and remove all */
/* unnecessary characters . Don't use this on widechar      */
/* strings or strings shorter than 10 characters           */
/* ***** */
```

只需要一个简单的命令, 用户就可以把丑陋的, 未格式化过的注释变成精心排列的文本.

Par 的手册页包含了许多例子.



读者很容易就可以把不同的 Par 命令映射到不同的按键上去, 通过这种办法, 可以为所有的文本, 注释, 线性表等设置格式化快捷键.

5.3.3 Tidy

如果用户从事网页开发, 或者 XML 文件编辑, 那么程序 `tidy` 就会成为你的好帮手. 这个程序可以清理代码, 使之符合 W3C 规范. 符合 W3C 规范指的是代码的构造必须遵守由 W3C 规定的 HTML 守则.

作为一个网页开发人员,笔者有时候会浏览别人写的 **HTML** 或 **XML** 文件,打开后只看到了一团乱麻.后来,笔者会在打开之前,先用 **tidy** 处理一下所有的 **.xml**, **.html** 或 **.html** 文件.这个工作可以借助 **Vim** 的自动命令来完成,自动命令可以添加在文件 **vimrc** 中.

对于 **XML** 文件,自动命令的具体代码是:

```
au FileType xml exe ":silent 1,$!tidy --input-xml true --indent yes -q"
```

对于 **HTML** 文件,自动命令是:

```
au FileType html exe ":silent 1,$!tidy --indent yes -q"
```

需要注意得是,这个命令会在打开文件时悄无声息地修改掉文件内容.在上面两个命令中,**Vim** 会在指定的路径中搜索程序 **tidy**,无论是 **Linux** 还是 **Windows**.

观察 **tidy** 的命令行参数后可以看到,它也可以用来重新缩进 **HTML/XML**.这个选项提高了文件的可读性,也更容易获取文件的概览.

因为 **tidy** 可以检查出文档的错误,所以用户可以把 **tidy** 的命令映射到一个键上,这样的话就可以随时检查文档是否符合 **W3C** 规范.



可以到 <http://tidy.sourceforge.net> 下载到最新版的 **tidy**.

139

5.4 小结

本章介绍了如何更好地格式化普通文本与代码.

首先,我们介绍了如何通过一对简单的 **Vim** 命令,把文本格式化成更易阅读的形式.还介绍了文本的对齐,还解释了纯文本编辑器在对齐方面的困难.接下来,我们创建了一个函数,该函数用于标记标题行,以及生成无编号与编号列表.我们还看到了 **Vim** 非常的灵活,比如用户可以告诉 **Vim** 是否需要对所选中的文本中的每一行执行操作,又或者是是否让用户来处理,并将待处理的行一次都反馈给用户的函数.

接下来介绍了在 **Vim** 中如何格式化代码,尤其是缩进.因为每个人都有着自己独特的编码风格,所以很难有一个通用的功能来完成代码的格式化. **Vim** 提供了一个灵活的接口,使得用户可以按照自己的需要来设置.我们还介绍了一些用来迅速格式化代码的技巧,甚至还包括如何格式化从其他地方粘贴到 **Vim** 中的代码.

最后,介绍了如何利用外部工具来使得 **Vim** 更加完美.外部工具可以帮助用户格式化文本与代码,我们介绍了几种比较流行的外部工具,以及在 **Vim** 中使用它们的方法.

到了这里,用户应该可以熟练地利用 **Vim** 内建功能,使得它更符合自己的需求.接下来,读者将学习到如何编写 **Vim** 脚本.

140

第六章 Vim 脚本基础

141

Vim 的一个最强大之处是允许高级用户通过编写脚本增强 Vim 的功能. 通过脚本, 用户几乎可以往 Vim 中添加任意功能, 而且很容易与其他用户分享.

这一章会介绍编写 Vim 脚本的基础知识, 讨论的主题包括:

- 编写语法高亮脚本
- 安装与使用脚本
- 不同类型的脚本
- 如何开发脚本
- Vim 脚本的基本语法
- 在 Vim 脚本中使用其他脚本语言

学习完这一章之后, 对于如何使用 Vim 脚本, 读者应该会有一个基本的概念, 而且有能力写出一个简单的脚本, 从而为 Vim 添加新的功能.

6.1 语法高亮方案

在许多程序员看来, 根据语法来高亮代码是 Vim 最重要的特性之一. 语法高亮不仅使代码看起来更清晰, 还可以帮助用户发现编码错误. Vim 语法高亮系统所使用的脚本非常像 Vim 脚本, 但是语法高亮脚本定义的是颜色, 而不是功能. 下一节介绍如何创建一个语法高亮方案.

142

6.1.1 第一个语法高亮文件

简单来说, 语法高亮的关键是识别出文本中的特定单词与结构, 然后再给它们设置上对应的颜色. 然而, 大部分情况下要稍微复杂一点, 因为语法高亮还要识别上下文语境. 假设现在要语法高亮下面的代码:

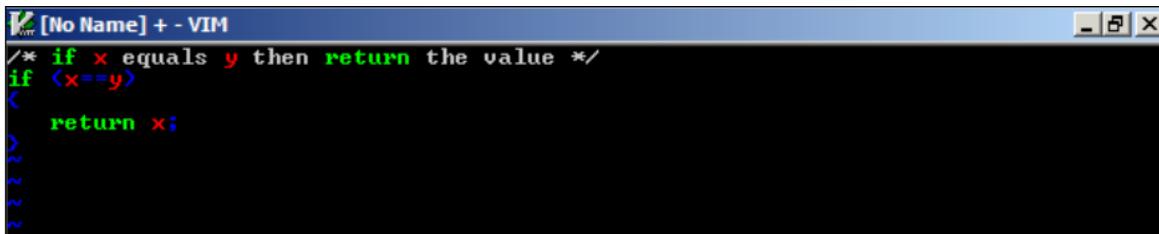
```
/* if x equals y then return the value */
if (x == y)
{
    return x;
}
```

如果仅仅是根据单词与符号来匹配, 那也可以得到一个相当不错的结果. 下面是具体的配置命令 (每个选项的意义已经在第二章中进行了介绍):

```

:syntax keyword myVars x y
:syntax match mySymbols "[{}() ;=]"
:syntax keyword myKeywords if return
:highlight myVars ctermfg=red guifg=red
:highlight mySymbols ctermfg=blue guifg=blue
:highlight myKeywords ctermfg=green guifg=green

```



从上图中可以看到, 代码部分的高亮还不错, 可是注释语句却不令人满意. 这是因为我们只是对单个单词进行匹配, 所以注释中的相同单词也会被匹配. 这种配置方法很难分辨出代码与注释.

143

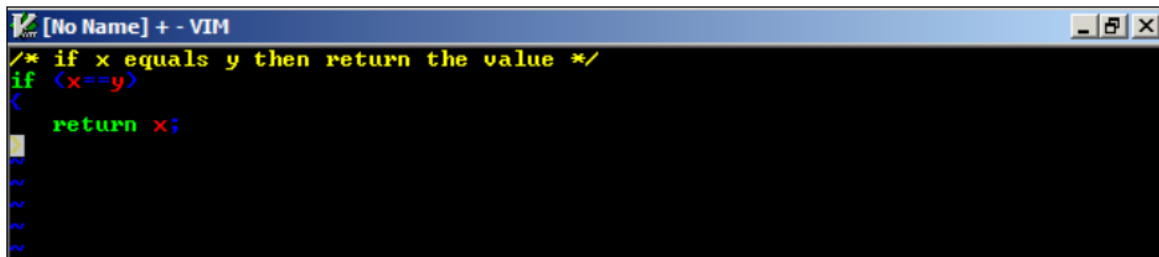
那么, 我们可以从这个简单的例子里学习哪些东西呢? 那就是, 与高亮比起来, 更重要的是要找到期望中的单词, 然后再给它们设置对应的颜色. 现在增加一些上下文的信息, 把 `/*` 与 `*/` 之间的部分标记成注释, 然后再高亮其余的部分. 代码部分被标记上颜色之后, 就不需要再上色了, 因此规则的顺序很重要. 具体的配置代码是:

```

:syntax match myComments "/*.**/"
:syntax keyword myVars x y
:syntax match mySymbols "[{}() ;=]"
:syntax keyword myKeywords if return
:highlight myVars ctermfg=red guifg=red
:highlight mySymbols ctermfg=blue guifg=blue
:highlight myKeywords ctermfg=green guifg=green
:highlight myComments ctermfg=yellow guifg=yellow

```

最终的效果是:



现在, 这段代码的高亮已经相当得体. 当然, 这只是一个例子, 而且只用到了 Vim 语法高亮的一小部分功能, 接下来将会介绍更多的内容.

6.2 区域高亮

前面的例子是用选项 `match` 来选择注释语句. 在某些情况下, 我们很难创建一个适当的匹配语句, 这时候, 就需要用到其他一些更方便的做法.

在 Vim 中, 用户可以选中一整个代码区, 然后再高亮它们, 为了选择一个代码区, 只需要提供区域的开始与结束. 对于前面的例子, 如果使用区域高亮的话, 具体的命令是:

```
:syntax region myComments start=/\/*\ end=\/*\//
```

144

有了这个命令, 很轻易就能匹配下面的任意一个注释块:

```
/* single line comment */
/*****
 * multi line comments
 *****/
/* multi line comment
 */
```

除了设置区域开始与结束, 选项 `region` 还可以做更多的事. 它还可以根据其他语法规则来高亮区域内的某些代码. 笔者常做的一个操作是为函数注释设置几个关键词, 比如 `FIXME`, `OBSOLETE`, `TODO`, 等等, 这样, 我可以把代码写成:

```
/* function: splitString()
 * args      : string
 * OBSOLETE
 */
function splitString(string) {
...
}
```

剩下的工作, 就是创建一个关键词组:

```
:syntax keyword myKeywords OBSOLETE FIXME TODO
```

现在, 需要修改命令 `region` 的设置, 以允许区域内包含其他语法元素, 修改后的命令是:

```
:syntax region myComments start=/\/*\ end=\/*\// contains=myKeywords
```

如果需要在区域内包含多于一个的语法组, 只需要把它们写成列表的形式 (列表元素之间用逗号分开), 再写到 `contains` 的后面.



只有当开始与结束都在同一行时, 你才能告诉 Vim 这个区域是正确的, 方法是在命令 `syntax` 中添加一行选项. 如果没有这个选项, Vim 会在遇到开始时就开始高亮代码, 遇到结束时 (或者是文件的末尾) 停止.

145

在某些情况下, 用户可能希望在一个区域中嵌套另一个区域, 对于这种情况, 用户必须把这项需求显式地告诉 Vim, 方法是在区域命令的末尾加上选项 `contained`:

```
:syntax region myComments start=/\/*\ end=\/*\// contains=myKeywords
contained
```

在某些情况下, 一个代码块可以出现在代码中的任意一个位置, 当然, 用户不想针对每一个代码块各写一个语法组, 这时候, 只需要把 `contains` 改成 `ALL`.

除了 `ALL`, 其他的关键词还包括:

- `ALLBUT`: 如果它是列表的第一项, 那么列表中的其余项目在区域中都不会被高亮显示

- CONTAINED: 如果该项在列表中, 那么带有选项 `contained` 的语法组就会在区域中被高亮显示
- TOP: 如果该项在列表中, 那么除了带有选项 `contained` 的语法组, 其他所有的语法组都会被包含进来

有了上面的帮助, 用户就可以轻易地选择大范围的语法组, 而不用一个一个地把它们写下来. 一个例子是选择除了 `myComments` 之外的所有语法组, 具体的命令是:

```
:syntax region myCodeblock start=/{/ end=}/{/ contains=ALLBUT,myComments
```



如果用户知道某些语法组经常一起使用, 可以把它们放在一个簇 (cluster) 中: `:syntax cluster myCluster contains=myKeywords,mySymbols,myConditions.` 引用一个簇时可以在名字前加 `@`: `:syntax region myComments start=/{/*/ end=/{*// contains=@myCluster.`

现在, 用户只需要把所有的配置命令都写到一个文件中, 再把这个文件放到 `VIMHOME` 的 `syntax` 子目录内. 文件的后缀名是 `.vim`, 前缀名是文件所对应的编程语言源代码文件的后缀名, 比如 C 语言源代码文件的后缀名是 `.c`, 那么它的语法文件就是 `c.vim`.

在前面的例子中, 所有语法组的名字都以 `my` 开始, 这是因为它所对应的编程语言是 `my` (当然, 只是假设性的). 如果是 C 语言, 那么比较好的做法是语法组的名字以 `c` 开始, 比如 `cKeywords`, `cConditions`, `cSymbols`, 等等).

继续前面的例子, 刚才说到 `my` 语法文件所对应的编程语言源代码文件的后缀名是 `.my`, 为了简便起见, 我希望 Vim 把文件识别成具有文件类型 `my`.

146

如果 Vim 无法识别用户正在编辑的文件, 那么就需要注册它们. 方法是在 `VIMHOME` 目录下的 `filetype.vim` 中添加几行. 如果该文件不存在, 可以手工创建. 对笔者的编程语言来说, 需要添加以下几行:

```
augroup filetypedetect
autocmd BufNewFile,BufRead *.my setfiletype my
augroup END
```

上面的代码告诉 Vim 把两行 `augroup` 之间的所有内容都添加给自动命令组 `filetypedetect`. 这个命令组用于确定文件所拥有的文件类型.

在我们的例子中, 命令的功能是无论何时打开一个后缀为 `.my` 的文件, 就把它的文件类型设置成 `my`. 如果用户还想区别其他文件类型, 还可以在两行 `augroup` 之间添加其他 `autocmd`.

注意, 如果 Vim 检查到用户正在编辑的文件的类型是 `my`, 它就会自动查找匹配的语法文件, 它会在目录 `VIMHOME/syntax/` 内查找以文件类型名开始的语法文件, 在这里就是 `my.vim`.

为了创建自己的语法文件, 用户所需要做的工作就是这些, 有了这些语法文件之后, 无论何时打开用户的文件, Vim 都会自动加载它们.



学习编写语法文件的最佳方法是阅读别人写的语法文件. 安装 Vim 后, 就已经安装了大量的语法文件, 涵盖了几乎所有常见的文件类型, 用户可以以它们为基础, 从而写出自己的语法文件.

另一方面, 如果用户想要在已存在的语法文件中添加额外的识别, 有两种方法可以做到. 其中一种是找到那个文件, 然后添加自己的内容, 另外一种更好的办法是使用 Vim 的后处理功能, 该功能可以覆盖 Vim 已经加载的脚

本或语法文件. 第二种办法可以做到无论系统上的脚本怎么更新, 都不用改动自己的部分, 因为它们已经和脚本隔开了.

使用后处理器的秘诀是脚本文件的存放位置. 在用户的 `VIMHOME` 目录下, 有一个子目录 `after` (如果不存在, 则手工创建). 无论 **Vim** 在何时查找脚本, 语法文件, 或配色方案, 它都会在 `after` 子目录内查找同名的文件. 比如说, 如果 **Vim** 找到了 `VIMHOME/syntax/c.vim`, 它就会接着查找 `VIMHOME/after/syntax/c.vim`, 看看是否有需要覆盖的地方. 这种情况同样适用于从下面这些目录找到的脚本:

147

- `plugin`
- `ftplugin`
- `indent`
- `autoload`
- `syntax`
- `colors`

只要用户需要某个文件, 就可以把上面的任意一个目录放到 `after` 目录下, 如果 **Vim** 找到了文件, 就会使用它.

6.2.1 配色方案与语法高亮

在前面的例子中, 通过命令 `:syntax` 添加了用户自己的高亮色彩组. 这使得用户对配色具有了完全的控制权, 但同时也限制了用户能使用的颜色的种类. 因此, 在其他地方可能就不能用配色方案所定义的颜色了.

一个更好的办法是使用 **Vim** 已经定义好了的色彩组, 这就把色彩定义与语法高亮分成了两个部分. 通过这种方法, 无论在何时修改了配色方案, 语法高亮都会自动地更新.

下面的命令可以列出所有已经定义了的颜色:

```
:highlight
```



或者, 用户还可以看一下配色方案文件, 这些文件存放在 `VIMHOME` 的 `colors` 目录下.

6.3 使用脚本

每个人都会有一些特定的编辑器需求, 其中有些需求比较简单, 比如按键绑定, 而有一些则比较复杂, 当然, **Vim** 不可能满足每个人的需要, 所以它提供了脚本供程序员扩展 **Vim**.

148

但是如果用户不是个程序员, 或者没有时间自己开发脚本, 那又该怎么办呢? 这当然不是个问题, **Vim** 是免费发布的, 因此有很多用户同样免费发布自己开发的脚本. 他们中的许多人会把自己的脚本放到 <http://www.vim.org> 上供人免费下载, 其他用户很容易就可以搜索到自己想要的脚本.

6.3.1 脚本类型

在 Vim 官网可以找到大量的脚本, 它们的功能各异, 有的很简单 (比如插入日期), 有的则比较复杂 (比如 IDE 编程环境), 但是实际上, Vim 只能识别某几种预定义的脚本类型组。

如果认真查看一下脚本类型, 就会发现它们可以分为两组, 第一组是全局插件组, 该组包含的脚本会在 Vim 启动时, 或者是在用户执行某些特定的函数调用时补初始化。这种脚本的典型例子包括为 Gvim 添加菜单, 为已经定义了函数添加功能, 又或者是根据用户的需要修改某些特性。

第二组是文件类型插件组。该组包含的脚本和某个 (或某些) 特定的文件类型相关联, 只有当相应类型的文件被打开或创建时, 才会加载脚本。组内脚本的功能可以是为某个特定的文件类型添加特性, 还可以是某些特定的工具。比如, 为某种编程语言的编译操作定义一个快捷键, 或者是在程序员编写的每一个函数上方添加一段注释。组内的脚本还包括语法高亮, 虽然和其他脚本相比, 语法高亮脚本的安装位置不太一样。

6.3.2 安装脚本

下载脚本后, 它们的格式通常是下面三种之一:

- 一个单一的 .vim 文件
- 一个压缩文件 (通常是 Zip 格式), 里面包含了一个或多个 .vim 文件, 以及文档
- Vimball 格式, 一种 Vim 脚本安装文件

如果待安装的脚本仅仅是一个单一的脚本文件, 安装的方法是把它复制到 VIMHOME/plugin 目录下, 如果是和某种文件类型相关的脚本, 就复制到 VIMHOME/ftplugin。如果用户使用的是多用户操作系统, 只需要把它们安装在 Vim 安装目录 (而非 VIMHOME) 下面的同名子目录中, 就可以同时为所有的用户提供服务。

149

如果脚本被打包成压缩文件, 其安装方法就很难说清楚。典型的安装方法是把压缩文件复制到 VIMHOME, 然后再解压。解压后, 压缩包中的文件会自动存放对应的目录内。通常在压缩文件中都有一个 README 或 INSTALL 文件, 它们详细介绍了脚本的安装方法。



如果脚本是在 <http://www.vim.org> 上找到的, 那就同时也能找到脚本的安装方法。

最后一种格式需要通过 Vimball 来安装, Vimball 是为 7 及以上版本的 Vim 而开发的。Vim 接收若干个文件, 然后把它们组合成一个单一的 Vim 脚本归档文件, 文件的后缀名是 .vba, 意思是 Vimball。

在开始使用 Vimball 之前, 得先安装 Vimball 脚本, 这个脚本用于读取与安装 Vimball 文件。和其他脚本一样, 用户可以到 Vim 官网下载到该脚本。



在 http://www.vim.org/scripts/script.php?script_id=1502 上可以下载到最新版的 Vimball。

安装完 Vimball 脚本后, 就可以用它来安装其他 Vim 脚本。

假设用户现在有一个名为 myscript.vba 的 Vimball 文件, 为了安装它, 先在 Vim 中打开该文件, 打开文件后, Vim 就会告诉用户如何安装 myscript.vba。安装的方法通常是执行下面的命令:

```
:source %
```

命令会把脚本安装到从选项 `runtimepath` 中找到的第一个目录内. 如果想修改安装目录, 就把命令换成:

```
:UseVimball PATH
```

用户需要把 `PATH` 替换成脚本的安装路径. 需要注意的是, 有些脚本只能安装在特定的目录下, 否则的话, 脚本就不能正常工作.

有时候, 用户可能想在安装之前查看一下 `Vimball` 中包含的文件. `Vim` 提供了一个这样的命令, 使用方法是在打开 `Vimball` 之后, 执行:

```
:VimballList
```

查看后, 如果没什么问题, 就可以用 `:source` 或 `:UseVimball` 安装.

150

6.3.3 卸载脚本

通常来说, 不存在用于卸载脚本的自动化方法, 用户必须手动地把文件删除掉. 不过 `Vimball` 含有卸载机制. 如果用户记得安装时所用的 `Vimball` 文件名, 就可以通过它来卸载脚本. 卸载 `Vimball` 的命令是:

```
:RmVimball vimballname
```

把 `vimballname` 替换成安装脚本时所用的 `Vimball` 文件名. 如果脚本不是安装在默认路径下 (通过 `:UseVimball` 命令), 就在命令的末尾加上安装路径:

```
:RmVimball vimballname path
```

为了把 `Vimball` 与将要删除的文件关联起来, 脚本会在 `VIMHOME` 下创建一个名为 `.VimballRecord` 的文件. 注意, 如果你把这个文件删除了, 就不能卸载之前所有的, 通过 `Vimball` 安装的脚本 (当然, 还可以通过手工删除来卸载).

6.4 脚本开发

在使用 `Vim` 的过程中难免会遇到这样的情况: 自己想要的功能, `Vim` 却不提供. 所以现在正好就是学习 `Vim` 脚本开发的好时机.

不过, 在开始前得先考虑几个问题. 首先, 用户必须确定自己所想要的功能其他人还没有为此写过脚本 — 干嘛要自己造轮子呢? 如果已经有人写了一个脚本, 而这个脚本的功能与你的非常接近, 那干嘛不直接修改他的脚本, 从而扩展它的功能, 使得它既能服务于你, 又能服务于他人? 这种开发方法可以缩短时间, 同时还可以避免类似脚本的泛滥.

如果找不到满意的脚本, 那就得自己开发. 对于这些情况, 首先要考虑当脚本开发完成后, 是否会发布它. `Bram Moolenaar` 将 `Vim` 免费发布, 很多脚本开发人员也是这么做的, 所以笔者也希望你能发扬分享的精神.

151



关于开源许可证的更多信息, 登陆 <http://www.opensource.org/>.

如果用户决定分享你的脚本, 那你最好早早地就考虑到 `Vim` 可以运行在多种不同的平台上, 所以用户的脚本最好也能够在这平台上运行, 这意味着:

- 不能期望某些外部工具是可用的

- 不能期望某些外部工具已经安装了,即使你的确安装了它们
- 用户必须记住,在不同的平台上,文件系统也有所不同
- 用户必须记住某些功能只能在某些平台上使用
- 尽量使得各个特性是可配置的,因为其他人可能并不喜欢它原来的样子

把这些记在心里后,接下来就可以学习开发 Vim 脚本,现在,先来看一些实际的例子.

6.4.1 脚本开发基础

接下来的几节将会介绍一些基本的类型与结构,知道这些是开发出好脚本的前提.

如果用户是一名程序员,那就会对 Vim 的脚本语言感到很熟悉,因为它的结构与其他编程语言是类似的.

152

类型

简单来说, Vim 只有两种类型的数据 — 字符串与数值. 之所以是简单来说,是因为这两种类型还包含了其他的子类型. 一个数值可以有三种表示方法:

- 十进制: 1, 2, 3, 100 等
- 十六进制: 0x01, 0x02, 0x03, 0xa0, 0x64 等
- 八进制: 01, 02, 03, 012, 0144 等

在表示十六进制与八进制数时,需要分别在数的左边加上 0x 与 0. Vim 可以方便地对数字进行运算,即使数字间的进制不太相同,比如,可以计算下面的表达式:

```
:echo 10 + 0x0A + 012
```

运算结果是 30.

Vim 中的字符串用一对单引号或双引号括起来,比如:

```
:echo "this is a string"
:echo 'this is a string'
```

如果想在字符串中表示用于括住字符串的字符(单引号或双引号),就用反斜杆转义:

```
:echo "this is a string with a \" double quotes"
:echo 'the double quote " does not need escaping here'
```

使用单引号还是双引号取决于具体的场景.

在单引号括住的字符串中,字符串将会按照字面显示,也被称为字面字符串. 这就意味着在单引号括住的字符串中不能使用特殊的转义字符.

比如,下面的字符串可以按照预想中的方式(分成两行显示)输出:

```
:echo "string with\n two lines"
```

但这个就不可以:

```
:echo 'string with\n two lines'
```

除了换行符 `\n`, 其他的转义字符序列还包括:

<code>\n</code>	换行符
<code>\r</code>	回车符
<code>\t</code>	制表符
<code>\123</code>	该八进制数所表示的字符
<code>\x123</code>	该十六进制数所表示的字符
<code>\u</code>	最多 4 个十六进制数所表示的字符
<code>\f</code>	换页
<code>\e</code>	转码
<code>\b</code>	退格

除了这些转义字符序列, 还可以插入其他 Vim 可识别的按键缩写, 比如 `<CR>`, 与 `<ESC>`, 但是得在前面加上反斜杆: `\<CR>`, 甚至还可以插入其他快捷键缩写, 比如 `<C-W>`, 但同样得加反斜杆.

变量

Vim 有 5 种类型的变量, 使用方法大不相同 (虽然定义的方式是相同的). 这 5 种类型是:

- 字符串: `"this is a string"` 就是一个简单的字符串
- 数值: 比如 123 或 `0x123`
- 线性表: 含有多个条目的有序序列 (或有序数组)
- 字典: 一个无序的关联数组, 保存有 键 -值 对
- 函数引用: 指向一个函数的引用

变量的名字可以包含字母, 数字与下划线, 但不能以数字开始.



尽量使用有意义的名字, 始终记住可能会有其他人阅读你的代码. 为了避免自己的变量名与别人的变量名混淆, 可以让自己的变量名以某个特殊的名字开始, 比如自己姓名的首字母缩写: `KSmyvariable`. 如果有多个程序员在开发同一套脚本, 那还可以用脚本文件名的缩写, 比如 Vim 排序脚本中的变量名可以是 `VSmyvariable` 或 `VSSmyvariable`.

所有类型的变量都是通过命令 `:let` 定义:

```
:let myvar = VALUE
```

命令中的 `VALUE` 取决于变量的具体类型. 对于字符串与数字, 定义的方式与前面讲过的相同, 比如:

```
:let mystringvar = "a string"
:let mynumbervar = 123
```

处理字符串与数值类型的变量时, 根据具体的使用方式, 这两种类型可以互相转换, 这意味着即使执行了:

```
:let mystringvar="123"
```

也可以使用:

```
:let mynumbervar=mystringvar-23
```

在算术表达式中,mystringvar 会自动转换成数值类型.



可以通过加 0, 从而把一个字符串强制转换成数值, 比如 :let mynumber=mystringvar+0. 为了把一个数值强制转换成字符串, 可以使用函数 string(): :let mystring=string(mynumber).

这种自动类型转换对线性表与字典不适用, 因为它们包含的值可能一点也不相同. 下面的表格汇总了类型转换的各种情况:

输入类型	结果类型
"hello" . "world"	"hello world" (字符串)
"number" .123	"number 123"(字符串)
"123" + 10	133 (数字)
"123" - 10 . "hits"	"113 hits" (字符串)
"123" - 10 + "hits"	113 (数值)

定义线性表的方式是将表内的各个值用逗号分开, 外面再包围一对中括号:

```
:let mylistvar1 = [1,2,"three",0x04,myfivevar]
```

线性表的元素还可以是一个线性表:

```
:let mylistvar2 = [[1,2,3],["four","five","six"]]
```

可以看到, 上面定义的线性表包含了字符串, 数值, 和线性表. 因此线性表可以作为存放不同类型变量的容器. 稍后将会介绍如何使用线性表中的变量, 以及如何与其他线性表一起工作. 创建一个字典变量的命令是:

```
:let mydictvar1 = {1: "one", 2: "two", 3: "three"}
```

命令创建了包含三个项目的字典变量, 项目的键是数值, 值是字符串.

不管把键写成数值, 还是字符串, Vim 都会把它们转换成字符串, 所以上面的命令会把键值对定义成 1:one. 字典中还可以包含其他字典, 比如:

```
:let mydictvar2 = {1: "one", 2: "two", "tens":{0: "ten", 1: "eleven"}}
```

可以看到, 键不需要遵守特别的顺序, 也不一定非得是数值类型 (比如例子中的 "tens").

稍后将会介绍如何访问字典中的值, 以及字典与线性表之间如何互相转换.

最后一种变量类型是函数引用, 这种类型的变量包含了指向某个函数的引用, 和其他类型的变量相比, 其不同点是它是可执行的. 定义函数引用的命令是:

```
:let Myfuncrefvar = function("Myfunction")
```

命令把变量 Myfuncrefvar 绑定到函数 Myfunction. 需要注意的是, 变量名以大写字母开始, 这是因为所有的用户自定义函数的函数名都以大写字母开始, 因此作为函数执行的变量也要遵命这个规定.

在使用函数引用类型的变量时, 只需要在函数名的后面加上一对括号:

```
:echo Myfuncrefvar()
```

除此之外, 还可以用:

```
:call Myfuncrefvar()
```

如果函数所绑定的函数需要输入参数, 那就把参数放到括号中, 就像 `Myfuncrefvar(arg1, arg2, ..., argN)`.

156

变量可以拥有不同的作用域, 这意味着某些变量只能在函数内访问, 而有些变量可以在任意一个地方被访问到.

脚本开发人员需要把变量的作用域告诉给 Vim, 方法是在变量名的前面加上作用域指示符.

如果在定义变量时没有定义作用域, 那就是全局的, 如果变量是在函数内定义的, 那么作用域仅限于函数内部. 总共有以下 8 种作用域:

- v: Vim 预定义的全局作用域
- g: 全局作用域
- b: 缓冲区作用域 — 只在定义变量的缓冲内有效
- t: 标签页作用域 — 只在定义变量的标签页内有效
- w: 窗口作用域 — 只在当前窗口内有效
- l: 函数作用域 — 只在定义它的函数内部有效
- s: 来源文件作用域 — 只在通过命令 `:source` 加载的文件内有效
- a: 参数作用域 — 专门用于函数的参数



Vim 脚本的注释以引号开始, 例如 `"this is a comment.`

下面的程序用到了几个不同的作用域:

```
let g:sum=0
function SumNumber(num1,num2)
    let l:sum = a:num1+a:num2
    "check if previous sum was lower than this
    if g:sum < l:sum
        let g:sum=l:sum
    endif
    return l:sum
endfunction
" test code, this will print 7 (value of l:sum)
echo SumNumbers(3,4)
" this should also print 7 (value of g:sum)
echo g:sum
```


虽然局部变量与全局变量的名字可以相同,但是如果用户已经知道某个全局变量的名字,最好就不要再使用同名的局部变量。



尽量使用具体的作用域. 这种方法可以避免全局变量被一些你无法控制的变量所污染.

条件

开发 Vim 脚本时,有时候在执行代码之前需要检查某个条件是否满足. 大多数现代编程语言使用条件语句进行条件检查. Vim 同样也有这种语句,它的格式是:

```
if condition
    code-to-execute-if-condition-is-met
endif
```

如果 condition 的值为真, if 和 endif 之间的语句就会执行, 否则就不执行.

那么, 在 if 结构中可以使用哪些条件? 主要有两种 — 使用逻辑运算符的语句, 与使用字符串运算符的语句. 现在来看一下这两种运算符如何使用. 其总的形式都是:

```
value1 OPERATOR value2
```

命令中的 OPERATOR 是用于比较 value1 与 value2 的运算符. 比如:

```
value1 >= value2
```

如果 value1 大于或等于 value2 则表达式为真. 这只是逻辑运算符中的一种, 其他的逻辑运算符还有:

- val1 == val2: 如果 val1 等于 val2, 则表达式为真
- val1 != val2: 如果 val1 不等于 val2, 则表达式为真
- val1 > val2: 如果 val1 大于 val2, 则表达式为真
- val1 < val2: 如果 val1 小于 val2, 则表达式为真
- val1 >= val2: 如果 val1 大于或等于 val2, 则表达式为真
- val1 <= val2: 如果 val1 小于或等于 val2, 则表达式为真

这些运算符既可以用于数值类型, 也可以用于字符串类型, 因为 Vim 可以自动进行转换. 如果是对字符串进行比较, 在比较时会逐个比较各个字母的 ASCII 码值. 例如 "bbb" > "aaa" 为真, 而 "abc" > "abd" 为假 (这是因为 a 的 ASCII 值比 d 小).

如果是对字符串进行处理, 还有更多的运算符可供选择. 如果用户想查看某个字符串是含有特定的子字符串或字符, 则可以使用部分匹配运算符, 使用方式是:

- str1 =~ str2: 如果 str1 包含 str2, 或 str1 与 str2 相同, 则为真
- str1 !~ str2: 如果 str1 不包含 str2, 并且 str1 与 str2 不相同, 则为真

使用这两个运算符时, `str2` 通常是一个模式, 并且可以使用 Vim 的正则表达式 (见 `:help regexp`). 这意味着不仅仅可以进行简单的匹配, 还可以是非常复杂高级的匹配.

所有的这些条件运算符都可以用于 `if` 语句内, 稍后你将会看到, 它们还可以用在其他地方.

现在来看几个条件表达式的具体使用示例.

在某些情况下, 用户可能希望某块代码只在某个条件满足时才执行, 当条件不满足时执行另一块代码. 这种情况下可以使用两个 `if` 条件 — 一个用于检查条件是否为真, 另一个用于检查条件是否为假. 不过, 还有另一种方法.

另一种方法是使用 `if-else-endif` 语句, 它的使用方法是:

```
if condition
    code-to-execute-if-condition-is-true
else
    code-to-execute-if-condition-is-NOT-true
endif
```

另一种情况是需要检查多个条件, 而且只有其中一个条件会满足, 代码的形式是:

```
if condition1
    code-to-execute-if-condition1-is-true
else
    if condition2
        code-to-execute-if-condition2-is-true
    endif
endif
```

可以看到, `condition1` 与 `condition2` 中只有一个的值可以为真, 并且两个都可以为假. 但是这样的代码是成簇的, 额外的 `endif` 如果放的位置不对, 会导致错误的 `if` 结构.

一个更好的方式是使用 `if-elseif-else` 结构:

```
if condition1
    code-to-execute-if-condition1-is-true
elseif condition2
    code-to-execute-if-condition2-is-true
endif
```

这段代码所做的工作与前一个的相同, 但是它的可读性更高. `elseif` 出现的次数可以多于一次, 这就意味着可以在同一个结构中判断多个条件, 而不用担心会出现什么问题.

稍后将会介绍如何在循环结构中使用条件判断.

使用线性表与字典

之前的内容已经介绍了如何创建线性表与字典, 现在讨论如何访问这两个结构中的数据.

如果用户已经有了一个线性表变量, 并且想要访问表中的某个变量, 访问的方法是在线性表变量的名字后面加上一对方括号, 并在括号内填上所要访问的值在线性表中的索引. 索引是表项在线性表中的存放位置, 从 0 开始. 假设我们想要访问下面线性表中的元素 `"three"`:

```
:let mylistvar1 = [1,2,"three",0x04,myfivevar]
```

"three" 的索引是 2, 所以可以这样访问它:

```
:echo mylistvar[2]
```

如果线性表中还有线性表, 比如 mylistvar2:

```
:let mylistvar2 = [[1,2,3],["four","five","six"]]
```

如果想要访问 "four" 所在的元素, 就需要访问里面那个线性表的索引 0, 与外面那个线性表的索引 1:

```
:echo mylistvar2[1][0]
```

在线性表中还可以使用负的索引值. 如果索引值是负的, 那个这个索引就是相对于表的末尾, 而不是开头. 所以, 访问 "four" 的另一种形式是:

```
:echo mylistvar2[-1][-3]
```

因为 -0 与 0 是相同的, 所以最后一项的索引是 -1.



如果试图访问一个不存在的元素, Vim 就会返回一个错误. 为了避免返回错误信息, 可以用 `get()` 获取元素: `:echo get(mylistvar1, 2)`, 其中 2 是试图访问的元素的索引.

如果想要往一个已存在的线性表中添加一个新元素, 有很多种方法都可以做到. 最简单的方式是用函数 `add()`, 它的一个使用示例是:

```
:let mylistvar3 = [1,2,3,4]
:call add(mylistvar3, 5)
:echo mylistvar3
```

上面的命令在 mylistvar3 添加一个新元素 5, 然后打印整个线性表. 添加元素的另一种方法是使用 Vim 的拼接功能. 运算符 `+` 可以用来拼接两个线性表, 它的一个使用示例是:

```
:let mylistvar4 = [1,2,3,4]
:let mylistvar4 = mylistvar4 + [5,6,7,8]
:echo mylistvar4
```

上面的命令先是创建一个具有 4 个元素的线性表, 然后再拼接上一个新的线性表, 再把拼接后的线性表赋给自己. 最终, 线性表中将含有 8 个元素.



在拼接两个线性表时, 还有一个更方便的运算符: `+=`. 这个运算符可以把右手边的线性表拼接到左手边的线性表上, 比如: `:let mylistvar4 += [5,6,7,8,]`

如果待拼接的线性表只含有一个元素, 那么它的效果与添加一个新元素是相同的.

除了运算符 `+`, 还可以用函数 `extend()` 来完成拼接. 它的一个使用例子是:

```
:let mylistvar5 = [1,2,3,4]
:call extend(mylistvar5, [5,6,7,8])
:echo mylistvar5
```

注意, `add()` 与 `extend()` 之间有很大的不同. 在上面的例子中如果把 `extend()` 换成 `add()`, 那么最终的结果是把一个线性表添加到另一个线性表中, 相当于:

```
:let mylistvar5 = [1,2,3,4,[5,6,7,8]]
```

这个线性表只有 5 个元素, 而且最后一个元素是一个含有 4 个元素的线性表.

移除元素的方法与添加元素的方法类似, 只不过要把 `add()` 换成 `remove()`, 比如:

```
:call remove(mylistvar5, 3)
```

这个命令从 `mylistvar5` 中移除索引为 3 的元素.

现在开始介绍如何访问与修改字典变量. 前面已经介绍过创建一个字典变量的命令类似于:

```
:let mydictvar1 = {1: "one", 2: "two", 3: "three"}
```

访问字典变量的方式与线性表非常类似. 假如用户想访问 "two", 那就得在字典变量的后面加个 `[]`, 并在括号内填上它的键值, 就像:

```
:echo mydictvar1[2]
```

看起来和访问线性表差不多, 但是如果把键值改成字符串, 那就不一样了:

```
:let mydictvar4 = {'banana': 'yellow', 'apple': 'green'}
```

现在, 用户想要访问苹果的颜色:

```
:echo mydictvar4['apple']
```

上面的命令会在屏幕打印出 'green'. 如果键值仅由 ASCII 字母, 数字, 与下划线组成, 则还可以用:

```
:echo mydictvar4.apple
```

键值的第一个字符必须是 ASCII 字母.

线性表中的元素是有序的, 并且含有索引, 相比之下, 字典是无序的, 键-值对中的键用于获取对应的值.

修改字典中的某个值的命令是:

```
:let mydictvar4['apple'] = 'red'
```

往字典变量中添加新元素的命令和上面的相同, 唯一需要注意的地方是所使用的键必须是变量中未出现过的.

用户可以在字典变量上附加一个函数, 使用该函数可以对字典变量中的元素进行一些操作, 让我们通过一个例子来说明.

比如说, 我们想要达到这样一个功能: 接收一个数值, 把数值中每一个数位上的数字转换成对应的英文拼写形式. 假设完成这个转换过程的函数是 `convert`, 而它所附加的字典变量是:

```
:let mynumbers = {0:'zero',1:'one',2:'two',3:'three',4:'four',
                  5:'five',6:'six',7:'seven',8:'eight',9:'nine'}
```

函数是:

```
function mynumbers.convert(numb) dict
    return join(map(split(a:numb,'\zs'),'get(self, v:val,"unknown")'))
endfunction
```

上面的函数代码中有两个地方需要注意.

第一个需要注意的地方是函数的定义方式与普通函数的定义方式相同, 除了函数名需要包含字典变量的名字, 并且在函数参数的后面需要加上关键字 `dict`. 这个关键词告诉 Vim 把该函数当成一个字典函数, 并对一个特殊的变量作操作 — `self`, 在这个例子中, 变量 `self` 指向函数所关联的字典变量. 这意味着我们可以通过 `self[1]` 获取值 "one", 以此类推. 函数的实现由四个子函数组成:

163

- `split`: 把存放在变量 `a:numb` 中的参数切分到一个匿名线性表中, 例如:

```
:let a = split("one two")
:echo a;      " this prints "one"
```

- `map`: 把一个给定的命令映射到线性表 (通过调用 `split` 得到的线性表) 中的每一个元素上, 例如:

```
:let mylist = ["one", "two", "three"]
:call map(mylist, "<" . v:val . ">")
:echo mylist[0] " this prints <one>
```

- `get`: 从 `self` 中取一个键值是 `v:val` (从 `map` 中获取的值) 的元素, 例如:

```
:let mylist2 = ["one", "two", "three"]
:echo get(mylist2, 2, "none") " prints three
:echo get(mylist2, 3, "none") " prints "none"
```

- `join`: 连接所有的, 由 `map-get` 返回的元素, 例如:

```
:let mylist3 = ["one", "two", "three"]
:let mystring = join(mylist3, "+")
:echo mystring " prints one+two+three"
```

更多的信息请参考 `:help split()`, `:help join()`, `:help map()`, `:help get()`.

说得更明白一点, 函数 `mynumbers.convert` 的功能是接收一个范围内的数字 (`a:numb`), 然后把它切分成单独的数字, 再把每个数字当成一个键去索引字典 (`mynumbers`, 或 `self`) 中对应的值, 把所有返回的值连接起来, 值与值之间用一个空格分开, 最后再返回这个字符串.

现在, 用户可以把字典变量当成一个转换器, 把数值转换成对应的英文形式, 使用方式是:

```
:echo mynumbers.convert(12345)
```

上面的命令将会打出 "one two three four five", 正好就是 12345 的英文表示形式. 不仅如此, 我们可以以此为基础, 联想出其他更多的功能.

164

循环

在处理线性表或字典类型的变量时, 经常需要遍历变量中的所有元素, 对于这种情况, 程序员通常会使用循环结构, Vim 同样也提供了这种结构.

Vim 提供了两种形式的循环:

- `for` 循环
- `while` 循环

for 循环 for 循环可以有多种构造方式, 其中最简单的一种是:

```
for var in ranges
    do-something
endfor
```

在上面的形式中, 循环依次遍历每一个元素, 每次迭代都会更新变量 myvar 的值, 一个例子是:

```
for myvar in range(1,10)
    echo myvar
endfor
```

上面的命令使用了函数 range() 来生成从 1 到 10 的整数, 循环从 1 开始遍历所有的数. 循环的每次迭代都会把变量 myvar 的值更新成当前从范围中所取得的值. 循环结束后, 将会打印出 1 到 10 这十个数字.

虽然上面的命令并没有用到线性表, 但是很容易就可以把函数替换成线性表, 换成线性表后的形式是:

```
for var in list
    do-something
endfor
```

上面的命令形式非常简单, 现在来看一个具体的使用例子:

```
let mylist = ['a','b','c','d','e','f','g','h','i','j','k']
for itemvar in mylist
    echo itemvar
endfor
```

命令执行时, 会陆续打印从 a 到 k 的所有字母. 用 for 循环遍历线性表的代码都遵循上面的形式.

如果想要循环打印字典变量中的元素, 则需要多做一点工作, 这是因为字典变量的元素与一个键关联, 而不是索引. 但是有一个函数可以帮助用户从字典变量中提取键, 然后就可以像处理线性表那样, 遍历所有的键, 这个函数的名称是 keys(). 先来看一下使用示例:

```
let mydict = {a:"apple", b:"banana", c:"citrus" }
for keyvar in keys(mydict)
    echo mydict[keyvar]
endfor
```

在上面的例子中, 从字典变量 mydict 中获取到一个包含了所有键的线性表, 然后遍历表中的每一个元素, 将元素的值放到变量 keyvar 中, 然后再用 keyvar 从字典变量中获取对应的值.

因为字典变量中的各个元素是无序的, 所以上面的命令可能会打印出 banana citrus apple. 有个函数可以对键进行排序, 从而得到有序的值, 这个函数是 sort:

```
let mydict = {a:"apple", b:"banana", c:"citrus" }
for keyvar in sort(keys(mydict))
    echo mydict[keyvar]
endfor
```

使用 sort 的前提是键是可排序的, 这对上面的例子来说并不成问题: a 排在 b 之前, 而 b 排序在 c 之前.



sort 还可以第二个参数, 这个参数是一个函数名, 这就使得用户可以提供自己的排序函数. 更多的信息与示例请参考 `:help sort()`.

166

while 循环 接下来讨论的是 while 循环. 正如它的名字所表现得那样, 当条件为真时执行循环. while 循环的基本结构是:

```
while condition
    execute-this-code
endwhile
```

只要条件为真, 就会循环执行 while 与 endwhile 之间的代码. 现在来看一个具体的例子:

```
let x=0
while x <= 5
    echo "x is now "x
    let x+=1
endwhile
```

上面的代码定义了一个变量 `x`, 其初始值为 0, 然后开始执行循环, 直到 `x` 大于 5 为止. 在每次迭代时, 代码都会打印 `x` 的值, 然后再递增 `x`, 代码执行的结果是:

```
x is now 0
x is now 1
x is now 2
x is now 3
x is now 4
x is now 5
```

使用 while 循环时, 通过一些特别的语句, 可以用到一些额外的特性.

第一个语句是 `break`, 它可以马上停止循环, 并将流程跳转到 `endwhile` 之后. 一个例子是:

```
let x = 0
let y = 1000
while x <= 1000
    let y -= 10
    if y <= x
        break
    endif
    let x += 1
endwhile
```

上面的代码创建了两个变量: `x` 与 `y`, 分别赋值为 0 与 1000. 然后开始 while 循环, `y` 在每次迭代时都递减 10, `x` 递增 1. 如果发现 `y` 小于等于 `x`, 就通过 `break` 语句马上跳出循环.

除了 `break`, 在 while 循环中还可以用 `continue`.

执行这个语句会跳过本次循环的余下语句, 然后马上开始下一次迭代.

167

一个例子是:

```
let x = 0
while x <= 5
    let x += 1
    if x == 2
        continue
    endif
    echo "x is now " x
endwhile
```

上面的代码把 `x` 从 1 递增到 5. 我们希望除了当 `x` 等于 2 时, 代码把 `x` 的每一个值打印出来. 所以, 如果 `x` 等于 2, 就通过 `continue` 语句马上结束掉当前循环, 开始下一次循环.

代码的输出是:

```
x is now 1
x is now 3
x is now 4
x is now 5
```

关于循环, 需要知道的就这么多了.



`break` 与 `continue` 还可以在 `for` 循环中使用, 其使用方法与在 `while` 循环中的一样.

创建函数

前面介绍的所有内容已经用了相当多的代码片段, 这些片段都被组合成 **Vim** 函数. 但我们还没有真正讨论过如何构造函数, 以及函数到底是什么.

从一个简单的函数开始讨论:

```
function Name(arg1, arg2, ...argN) keyword
    code-to-execute-when-function-is-called
endfunction
```

`Name` 是函数的名字, 函数名以字母开始, 并且只能包含字母, 数字, 下划线.

`arg1` 到 `argN` 是函数的参数, 它们是调用函数时必须提供的东西. 如果不需要提供参数, 那就留空 (在函数名之后填一对空的括号: `()`). 函数最多可以有 20 个参数, 可以按照自己的喜好给这些参数命名.

在参数列表的右括号的后面可能会有一个关键词, 这个关键词告诉 **Vim** 该函数的用途, 以及应该如何调用该函数. 函数的关键词包括:

- `dict`: 告诉 **Vim** 该函数绑定到一个字典 (本章的前面出现过这个关键词)
- `range`: 告诉 **Vim** 函数为一个范围内的行调用一次, 而不是为每一行调用一次 (第五章出现过这个关键词)

在大部分情况下都不需要这两个关键字, 因此可以不管它们。

在函数内部, 是该函数被调用时所执行的代码。函数内部所有变量的作用域都是局部的, 当函数执行结束后就不能再被访问。如果用户希望在函数内部仍然可以访问某个外部变量, 或者把它当作一个参数传递给函数, 或者在函数名的前面加上 `g:`, 把它的作用域定义成全局的。

169



如果把一个变量作为参数输送给函数, 在传递参数时, 只是把参数的值传递给函数, 如果参数在函数内部被修改了, 并不会影响到函数外部的实际参数。

函数参数的作用域是 `a:`。

这里展示一个简单的函数, 它需要两个参数, 然后打印它们的和:

```
function PrintSum(num1, num2)
    let sum = a:num1 + a:num2
    echo "the sum is ".sum
endfunction
```

上面的代码展示了如何使用参数, 不过用户可能还想在函数内部更新作为全局变量的和, 就像下面这样:

```
let sum = 0
function PrintSum(num1, num2)
    let sum = a:num1 + a:num2
    echo "the sum is ".sum
    let g:sum = sum
endfunction
```

这样的话, 即使是在函数调用结束后, 仍然可以访问变量 `sum`。假设用户执行了下面的代码:

```
let sum = 1
call PrintSum(4,5)
echo sum
```

上面的代码最终会打印出 9, 因为 `PrintSum` 会修改全局变量 `sum`。

然而, 在函数内部直接修改全局变量并不是一个好的编程风格, Vim 提供了另一种修改全局变量的方法 — `return` 语句。

170

`return` 语句使函数停止执行, 并返回一个值。返回的值可以直接赋值给变量, 现在用 `return` 语句来修改上面的例子:

```
function PrintSum(num1, num2)
    let sum = a:num1 + a:num2
    echo "the sum is " sum
    return sum
endfunction
```

然后, 调用函数的代码改成:

```
let sum = PrintSum(4,5)
echo sum
```

如果在 `return` 语句之后还有其他语句, 那么这些语句永远也不会执行, 这是因为函数在执行完 `return` 语句后就会马止结束. 这也就是说每次只能返回一个值.



函数内可以有多个 `return` 语句, 但函数在遇到第一个 `return` 语句时就会返回. 有些人认为在一个函数内出现多个返回点是一种不好的编程风格, 除非你没有其他办法可以选择.

可变参数列表 在前面的例子中, 函数只有两个参数. 如果待求和的参数个数多于两个的话, 那又该怎么办呢? Vim 提供了可变参数列表, 方法是把 `...` 作为函数的最后一个参数.

现在把 `sum` 改写成尽可能多地吸收参数 (但至少应该提供两个参数):

```
function PrintSum(num1, num2,...)
    let sum = a:num1 + a:num2
    let argnum = 1
    while argnum <= a:0
        let sum += a:{argnum}
        let argnum += 1
    endwhile
    echo "the sum is " sum
    return sum
endfunction
```

171

上面这个新的函数引进了一些新的变量.

`argnum` 是一个计数器, 用它遍历 `num1` 与 `num2` 之后的所有参数.

函数参数的个数存放在一个特殊的变量中 — `a:0`, 所以用它来判断应该在什么时候停止循环.

为了访问到每一个变量, 把 `argnum` 当作索引, 用于访问可选参数列表, 访问的方式是使用变量 `a:{argnum}`. 用户可以把 `a:{}` 看作是由可选参数组成的线性表, 而 `argnum` 是线性表的索引.

对于参数列表中的每一个额外的参数, 都把它加到变量 `sum` 中, 当函数执行结束时, 打印 `sum` 的值, 并把它返回.

于是, 可以这样调用函数:

```
let sum = 0
let sum = PrintSum(4,5,6)
echo sum
let sum = PrintSum(4,5,6,5,4,3,2,1)
echo sum
let sum = PrintSum(1234,5432,3333)
echo sum
```

上面代码的执行结果是:

```
15
30
9999
```

用户可能希望把所有的参数当作一个线性表来访问, Vim 提供了这个功能, 这个特殊的变量是 `a:000`, 它的类型是线性表. 通过这个变量, 可以把函数 `PrintSum` 改写成:

```
function PrintSum(num1, num2,...)
    let sum = a:num1 + a:num2
    for arg in a:000
        let sum += arg
    endfor
    echo "the sum is " sum
    return sum
endfunction
```

这次, 通过 `for` 循环, 把可选参数一个接一个地传递给 `arg`, 然后再把变量 `arg` 中的值累加到 `sum` 上.

如果用户不再想保留某个函数, 可以用下面的命令把它删除:

```
:delfunction function-name
```

其中, `function-name` 是用户想要删除的函数的名字.



如果想看某个函数的具体实现, 可以用命令 `:function`. 例如, 对于前面的例子, 可以输入 `:function PrintSum`. 如果在 `:function` 之后没有跟上参数, 命令就会列出全部可用的函数.

除了自己定义的函数, Vim 还提供了许多预定义的函数, 它们可以用来完成各种不同的任务. 可以用下面的命令获取各个函数的有关信息:

```
:help 'function-list'
```

6.5 小结

这一章主要是为那些想通过脚本来扩展 Vim 功能的人而写. 本章开始先介绍了一种最简单的 Vim 脚本类型——语法高亮方案. 我们学习到了如何为编程语言 (或其他语法内容) 创建一个语法高亮文件, 从而得到更好的高亮效果.

之后, 讨论如何使用脚本文件, 介绍了几种 Vim 可用的脚本类型, 以及在哪里可以获取到它们. 在这一节结束时, 介绍了脚本的安装方法与卸载方法.

既然已经知道了如何安装以及如何使用脚本, 是时候介绍如何开发 Vim 脚本.

首先介绍了 Vim 脚本的基本变量类型, Vim 脚本变量几乎是无类型的 (只有两种基本类型). 然而, 我们仍然可以利用这些类型存放各种各样的数据.

在介绍完 Vim 脚本语言的基本结构之后, 开始介绍如何创建函数. 通过一个关键词, Vim 就会知道应该如何调用函数. 这就允许我们将函数绑定到某种特定的变量上, 比如字典变量. 于是, 我们就能定义一些可以直接对字典变量进行操作的函数.

在学习完这一章之后, 下一章将会介绍如何运用本章所学的知识, 写出一个结构良好的脚本.

172

173

第七章 Vim 脚本进阶

175

在前面一章, 我们已经学习了开发 Vim 脚本的基础知识, 现在, 将要把前面所学的知识融会贯通, 按照结构化的方法把它们组织起来, 并对脚本进行测试.

这一章涵盖的主题包括:

- 如何组织 Vim 脚本的结构
- Vim 脚本开发的一些技巧
- 如何调试 Vim 脚本
- 如何在 Vim 脚本中使用其他脚本语言

阅读完这一章之后, 读者将有能力运用 Vim 脚本语言与其他脚本语言开发出自己的脚本. 也就是说, 读者将有能力扩展 Vim 的功能.

7.1 脚本结构

前面已经介绍了 Vim 脚本的各个要素, 现在需要知道如何把它们组织在一起, 从而形成一个完成的脚本.

在大部分情况下, Vim 脚本仅由单个文件组成, 因此这一章的示例也仅限于单个文件. 我们还打算让其他人能够获取到脚本, 因此需要保证代码的可读性.

下面的几节将会逐个介绍一个格式良好的脚本的各个要素.

176

7.1.1 脚本头部

一个脚本文件在开头最好配上一个头部信息, 用来写明该脚本的用途. 头部应该包含下面这些信息:

- 脚本的维护人员
- 最后一次更新的版本
- 发布许可证 (最重要的信息)

一个示例是:

```
" myscript.vim : Example script to show how a script is structured.
" Version      : 1.0.5
" Maintainer   : Kim Schulz<kim@schulz.dk>
" Last modified : 01/01/2007
" License      : This script is released under the Vim License.
```

注意头部信息都是以 " 开始的, 也就是说它们都是一些注释.

头部还可以包含其他信息, 比如脚本可能依赖于其他脚本, 或者是该脚本对 Vim 版本的要求.

7.1.2 脚本加载检查

一种良好的编程习惯是检查脚本是否已被加载, 如果是, 则在继续其他操作之前, 先执行卸载操作. 这是因为脚本不仅会安装在系统的全局目录中, 还会安装在用户的 VIMHOME 目录下.

检查脚本是否已加载的函数可以这样实现:

```
if exists("loaded_myscript")
    finish "stop loading the script
endif
let loaded_myscript = 1
```

如果脚本未被加载, if 条件判断为假, 函数把变量 loaded_myscript 设置成非零.

当下一次加载脚本时, if 条件判断为真, 因为此时变量 loaded_myscript 已经存在, 然后函数停止加载脚本.

在某些情况下, 停止加载脚本可能并不是最好的做法, 因为用户可能修改了 VIMHOME 目录中的脚本版本. 所以, 这时候应该先卸载脚本, 然后再重新加载. 完成这项功能的函数可以这样实现:

```
if exists("loaded_myscript")
    delfunction MyglobalfunctionB
    delfunction MyglobalFunctionC
endif
let loaded_myscript = 1
```

脚本开发人员无法知道 Vim/vi 当前是否处于兼容模式下 (如果是 vi 的话, 则比较有可能), 所以比较好的做法是在脚本中保存编辑器的兼容模式. 这样做就可以确保脚本可以正常地使用 Vim 特定的功能. 把下面的代码加到加载检查语句的后面:

```
:let s:global_cpo = &cpo "store current compatible-mode
                        " in local variable
:set cpo&vim           " go into nocompatible-mode
```

最后在脚本的末尾恢复原来的兼容模式:

```
:let &cpo = s:global_cpo
```

7.1.3 脚本配置

用户在阅读别人开发的脚本时, 总是从头开始看起, 所以最好把所有的配置选项都放在脚本的开始. 配置选项可以是外部程序的路径, 脚本依赖的特定文件的名称, 文件类型等.

用户可能会在他的 vimrc 文件中改变 Vim 的配置, 所以开发人员需要确保脚本不会覆盖掉他原来的配置. 方法是事先检查配置是否已被设置过, 只有在没有设置过的情况下才设置它.

脚本中的设置语句可以这样写:

```

" variable myscript_path
if !exists("myscript_path")
    let s:vimhomepath = split(&runtimepath, ',')
    let s:myscript_path = s:vimhomepath[0]."/plugin/myscript.vim"
else
    let s:myscript_path = myscript_path
    unlet myscript_path
endif

" variable myscript_indent
if !exists("myscript_indent")
    let s:myscript_indent = 4
else
    let s:myscript_indent = myscript_indent
    unlet myscript_indent
endif

```

178

上面的例子设置了两个配置变量 — `myscript_path` 与 `myscript_indent`。先变量是否已存在, 如果不存在, 则在脚本的作用域内设置变量的默认值 (比如, `s:myscript_path`)。

如果用户已经设置了变量, 那么变量的值就赋给脚本作用域内的同名变量。

最终, 用户定义的变量用 `unlet` 移除, 这样的话, 它就不对全局作用域产生影响 — 配置只需要在脚本内起作用即可。

7.1.4 按键映射

如果需要的话, 还可以添加按键映射, 这些映射可以是函数调用, 变量设置等。和配置变量一样, 在设置某个映射之前, 需要检查一下该映射是否已经建立好了, 检查的语句可以这样写:

```

if !hasmapto('<Plug>MyscriptMyfunctionA')
    map <unique> <Leader>a <Plug>MyscriptMyfunctionA
endif

```

在上面的代码中包含了一些以前没有介绍过的东西:

- `hasmapto()`: 用于检查某个函数映射是否存在的函数
- `<unique>`: 如果存在相同的映射存在, 则报错。
- `<Leader>`: 由用户决定使用哪个映射前导字符。 `<Leader>` 将会被全局变量 `mapleader` 的值所替换。
- `<Plug>`: 为一个函数建立一个唯一的全局标识符, 这样的话, 它就不会与全局作用域中的其他函数产生冲突。

把这些元素都组织在一起之后, 创建一个脚本, 用来检查是否存在某个映射, 已经绑定到唯一的函数标识 `<Plug>MyscriptMyfunctionA` 上。如果这样的映射不存在, 就把 `<Leader>a` 映射到标识符上 — 除非 `<Leader>a` 已经被其他人占用了, 此时 Vim 就会报错。

179

用户可能好奇 Vim 是如何从 `<Plug>MyscriptMyfunctionA` 得到 `MyfunctionA()` 的. 为此, 还需要建立其他一些映射:

```
noremap <unique> <script> <Plug>MyscriptMyfunctionA <SID>MyfunctionA
noremap <SID>MyfunctionA :call <SID>MyfunctionA()<CR>
```

第一个命令把 `<Plug>MyscriptMyfunctionA` 映射到 `<SID>MyfunctionA` 上. 代码中用到了 `<SID>`, 这是因为这个小标签会被 Vim 为当前脚本所生成的唯一 ID 给替换掉. 如果想制作一个全局的函数映射, 并且只在当前脚本作用域内可用 (例如 `s:MyfunctionA()`), 就需要这样的技术.

第二个命令把真正的函数 (`<SID>MyfunctionA()`, 也就是 `s:MyfunctionA()`) 绑定到 `<SID>MyfunctionA`.

设置完毕后, 当用户按下 `\a` (把 `mapleader` 设置成 `\`), 第一个映射把它翻译成 `<Plug>MyscriptMyfunctionA`, 它在脚本内定义, 因此 `<SID>` 的值是正确的. 因此, `<Plug>MyscriptMyfunctionA` 再次被翻译成 `<SID>MyfunctionA`, 最终被映射到调用本地函数 `s:MyfunctionA()`.

读者可能觉得上面所说的有点复杂, 而且 `MyfunctionA()` 可能本来就是一个全局唯一的函数. 但是, 如果函数名是一些常见的名字, 比如 `Add()`, `Delete()`, `Convert()` 等, 其他脚本很可能也实现了这些函数. 在这些情况下, 这些冲突的函数名会让 Vim 无法判断到底应该使用哪个函数. 当然, 用户可以给自己的函数取一些怪异的名字, 从而避免重名, 但是这种做法会让自己的代码变得杂乱无章, 而且还会污染全局作用域.

更多的信息请参考:



```
:help <SID>
:help <Plug>
:help 'script-local'
```

180

7.1.5 函数

函数几乎可以算作脚本开发过程中最重要的部分. 我们已经看到了如何创建一个函数, 而且已经注意到, 作用域标记 `s:` 把函数的作用域限定在脚本内, 通常要比全局范围内可见要好一点. 函数的一个例子是:

```
" this is our local function with a mapping
function s:MyfunctionA()
    echo "this is the script-scope function MyfunctionA speaking"
endfunction

" this is a global function which can be called by anyone
function MyglobalfunctionB()
    echo "Hello from the global-scope function myglobalfunctionB"
endfunction

" this is another global function which can be called by anyone
function MyglobalfunctionC()
    echo "Hello from MyglobalfunctionC() now calling locally:"
    call <SID>MyfunctionA()
endfunction
```

第一个函数是私有函数, 只有在脚本内可见, 另外两个函数的作用域是全局的. 需要注意的是, 因为全局函数知道当前脚本的 <SID>, 所以它们可以调用本地函数.

7.1.6 一个完整的脚本

如果读者已经看过前面几节, 那么现在读者手上就已经具备了构成一个完整的脚本的全部要素. 现在把所有的要素都集中在一起, 看一下一个完整的脚本是什么样子的:

```
" myscript.vim - Example script to show how a script is structured.
" Version      : 1.0.5
" Maintainer   : Kim Schulz<kim@schulz.dk>
" Last modified: 01/01/2007
" License      : This script is released under the Vim License.

" check if script is already loaded
if exists("loaded_myscript")
    finish "stop loading the script
endif
let loaded_myscript=1

    let s:global_cpo = &cpo "store compatible-mode in local variable
    set cpo&vim           " go into nocompatible-mode
" ##### CONFIGURATION #####
" variable myscript_path
if !exists("myscript_path")
    let s:vimhomepath = split(&runtimpath, ',')
    let s:myscript_path = s:vimhomepath[0]."/plugin/myscript.vim"
else
    let s:myscript_path = myscript_path
    unlet myscript_path
endif

" variable myscript_indent
if !exists("myscript_indent")
    let s:myscript_indent = 4
else
    let s:myscript_indent = myscript_indent
    unlet myscript_indent
endif

" ##### FUNCTIONS #####
```



```

" this is our local function with a mapping
function s:MyfunctionA()
    echo "This is the script-scope function MyfunctionA speaking"
endfunction

" this is a global function which can be called by anyone
function MyglobalfunctionB()
    echo "Hello from the global-scope function myglobalfunctionB"
endfunction

" this is another global function which can be called by anyone
function MyglobalfunctionC()
    echo "Hello from MyglobalfunctionC() now calling locally:"
    call <SID>MyfunctionA()
endfunction

" return to the users own compatible-mode setting
:let &cpo = s:global_cpo

```

现在, 读者手上就已经有了一个完整的脚本, 这是我们的第一个 Vim 脚本插件. 虽然它的功能还不是很丰富, 但是它已经展现了一个 Vim 脚本的完整结构. Vim 还有其他几种类型的脚本, 比如文件类型插件, 编译器插件, 和库函数脚本, 关于如何编写这些脚本, 可以参考:

```

:help 'write-filetype-plugin'
:help 'write-compiler-plugin'
:help 'write-library-script'

```



在 Vim 官网 <http://www.vim.org> 上可以找到大量的脚本, 读者可以从中获取灵感. 其中一些脚本属于库函数, 它们可以加快脚本的开发过程.

7.2 脚本开发技术

这一节将会介绍一些脚本开发过程中的小技巧. 有些技巧非常简单, 例如可以直接插入到脚本的一小段代码, 还有一些则非常值得一学.

7.2.1 Gvim 或 Vim

有些脚本在 Gvim 中使用时会有一些额外的特性. 包括添加菜单, 工具栏, 或者是其他一些只能在 Gvim 使用的功能. 那么, 开发人员应该如何知道用户现在是工作在 Vim 还是 Gvim 下呢? 其实 Vim 已经提前准备好了这些信息, 开发人员所要做的仅仅是检查特性 `gui_running` 是否已经开启. 为了完成这个操作, 需要使用一个函数 `has()`, 如果指定的特性是支持的, 则返回 1, 否则的话, 为 0.

一个例子是:

```
if has("gui_running")
    "execute gui-only commands here.
endif
```

这就是用来检查用户使用的是 Gvim 还是 Vim 的所有代码。需要注意的是, 仅仅检查特性 gui 是否开启是不完整的, 因为如果 Vim 在编译时开启了 GUI 选项, has("gui") 就会返回 1 — 即使当前并没有使用到图形界面。

183



执行 :help 'feature-list', 可以查看 has() 支持的其他特性。

7.2.2 操作系统类型

如果脚本将要在多种不同的操作系统中运行, 比如 Microsoft Windows, Linux, 读者将会发现需要处理许多问题。

需要处理的问题包括程序的存放位置, 程序是否可用, 以及文件的访问权限。

有时候, 操作系统还会影响脚本的结构, 因为脚本可能会调用外部工具, 或访问依赖于操作系统的功能。

Vim 允许脚本开发人员检查操作系统的类型, 这样的话, 脚本就可以根据操作系统的类型作出相应的动作, 比如停止运行, 或进行特定的设置。示例代码如下:

```
if has("win16") || has("win32") || has("win64") || has("win95")
    " do windows things here
elseif has("unix")
    " do linux/unix things here
endif
```

上面的示例只展示了如何检查 Windows 与 Linux/Unix。除了这些, 还可以检查其他类型的操作系统, Vim 支持的操作系统类型可以用下面的命令找到:

```
:help 'feature-list'
```

7.2.3 Vim 的版本

在最近这二十年, Vim 的快速发展添加了许多新功能, 有时候, 开发人员会在脚本中使用最新的函数, 因为它们使用起来最方便。但是如果用户的 Vim 版本比较老, 没有提供相应的函数, 那又该怎么办呢? 面对这个问题有三种做法:

184

1. 什么都不管, 让用户烦恼去吧 (不是个好主意)
2. 检查用户所用的 Vim 是否是旧版, 如果是的话, 就停止加载脚本
3. 检查用户所用的 Vim 是否过旧, 如果是的话, 就执行备用代码

第一个选择实在不是个好主意, 不推荐任何人使用它。

如果脚本无法处理旧版 Vim, 那么第二个选择是可接受的。可是, 如果对于旧版有备选方案可供选择, 那就最好用上。

现在讨论如何检查 Vim 的版本。

在检查 Vim 的版本之前,得先看一下版本号的结构.

版本号由三部分组成:

- 主版本号 (比如 Vim 7.2 中的 7)
- 次版本号 (比如 Vim 7.2 中的 2)
- 补丁号 (比如 Vim 7.2.103 中的 103)

前两个数字是真正的版本号,当某个补丁或小的特性被应用到 Vim 的某个版本上时,只有补丁号会发生变化.只有当修改足够多时,次版本号才会发生变化,只有当 Vim 的主要部分发生变化时,主版本号才会发生变化.

因此,如果需要检查用户所使用的 Vim 版本,就要检查这三个数字. 检查的示例代码是:

```
if v:version >= 702 || v:version == 701 && has("patch123")
    " code here is only for version 7.1 with patch 123
    " and version 7.2 and above
endif
```

if 的第一个判断是检查 Vim 的版本是否大于或等于 7.2 (注意,如果次版本号小于 10,就在前补 0). 如果第一个判断不满足,就检查版本号是不是 7.1,并且打上了编号为 123 的补丁. 如果补丁号大于或等于 124,那就表示补丁 123 已经应用在了 Vim 上.

185

7.2.4 打印很长的行

Vim 最初是用在比较老式的文本终端上,这些终端每一行的长度都不会超过某个限定值,如今大多数的终端已经不再限制一行的长度,但是这个限制在 Vim 中仍然时有出现.

如果使用 echo 语句往屏幕打印比较长的行时,这个限制就会对打印行为产生影响.即使运行 Vim 的终端宽度大于 80 个字符,Vim 仍然会在打印完 80 个字符后提醒用户按下回车键,然后接着打印后面的字符.有一个办法可以解决这个问题,使得在回显字符时,能够用上全部的终端宽度. Vim 窗口列数减 1 后就是窗口的宽度,比如,如果 Vim 窗口的列数是 120 个字符,那么窗口的宽度就是 119 个字符.

下面的函数用于打印长度为屏幕宽度的一行:

```
" WideMsg() prints [long] message up to (&columns-1) length
function! WideMsg(msg)
    let x=&ruler | let y=&showcmd
    set noruler noshowcmd
    redraw
    echo a:msg
    let &ruler=x | let &showcmd=y
endfunction
```



这个函数最早出现在 Yakov Lerner 所开发的 Vim 脚本中,该脚本可以在 <http://www.vim.org> 上找到.

现在,如果用户需要在脚本中回显一行比较长的行,可以使用函数 WideMsg(),使用的方式是:

```
:call WideMsg("This should be a very long line of text")
```

一行的长度仍然受到限制,但上限值并非原来的 79 个字符,而是 Vim 窗口的宽度。

186

7.3 调试 Vim 脚本

有时候,脚本并不会按照开发人员期望中的那样工作,在这种情况下,开发人员就得知道如何调试 Vim 脚本。这一节会介绍几种调试错误的方法。



结构良好的脚本拥有更少的错误,也更容易调试。

Vim 提供了一种用于调试的特殊模式,根据目标的不同,启动该模式的方法也有所不同。

如果 Vim 抛出了一些错误(在 Vim 窗口的底部打印它们),但是开发人员并不确定发生错误的地方,也不知道为什么会发生这些错误,这时候可以以调试方式直接启动 Vim。方式是带上参数 -D:

```
vim -D somefile.txt
```

当 Vim 开始读取第一个 vimrc 文件(大部分情况下是安装路径中的全局 vimrc 文件)时启动调试模式。

另一种需要进入调试模式的情况是开发人员已经知道某个函数存在错误,因此他只想调试这个函数。对于这种情况可以以普通方式启动 Vim(必要的话,还要加载包含待调试函数的脚本),然后执行:

```
:debug call Myfunction()
```

:debug 后面的所有东西都是待调试的内容。在上面的代码中只是调试函数 Myfunction() 的调用,除此之外,还可以写成:

```
:debug read somefile.txt
:debug nmap ,a :call Myfunction() <CR>
:debug help :debug
```

接下来看一下在调试模式下,我们可以执行哪些操作。

当执行到第一行需要调试的代码时,Vim 就停止加载,显示一些类似于下面的信息:

```
Entering Debug mode. Type "cont" to continue
cmd: call MyFunction()
>
```

进入到 Vim 脚本调试器后,可以对 Vim 发出一些指令,告诉它接下来如何操作。



如果用户对调试技术不是很熟悉,最好在开始调试脚本之前先把这一节看完。

在调试器内可以使用下面的命令(括号里面的是缩写形式):

- cont (c): 按照正常方式继续执行脚本/命令,直到下一个断点
- quit (q): 马上退出调试过程

187

- `interrupt (i)`: 停止当前过程, 就像 `quit` 那样, 但是回到调试模式中
- `step (s)`: 执行下一行代码, 执行完毕后回到调试模式. 如果下一行是函数调用或执行另一个文件中的命令, 则单步跟踪函数调用/命令执行
- `next (n)`: 执行下一行代码, 执行完毕后回到调试模式, 如果下一行是函数调用, 则会在函数返回后再回到调试模式
- `finish (f)`: 继续执行, 即使遇到断点也不停止, 执行结束后回到调试模式.

通过这些命令, 开发人员就可以知道脚本/函数如何执行. 如果想要重复运行上一次运行的命令, 只需要按下回车键.

如果需要的话, 可以在任意时刻执行另一个 Ex 命令 (参考 `:help 'ex-command-index'`^①), 但是需要注意的是, 在调试器内无法直接访问变量, 除非它们是全局的.

有时候, 开发人员想要查看的地方在很多行代码的后面, 而他不想一步一步地执行代码, 那样需要花很长的时间, 也非常繁琐.

188

对于这种情况, 开发人员可以在需要停止的地方插入一个断点, 然后在开始时执行 `cont`. 插入断点的命令有以下三种, 具体使用哪一种取决于开发人员的实际需求:

```
:breakadd func linenumber functionname
:breakadd file linenumber filename
:breakadd here
```

第一个命令把断点插入在特定的函数入口. 函数名 `functionname` 可以是模式, 比如 `Myfunction*`, 此时断点会插入在所有的, 其函数名以 `MyFunction` 开始的函数的入口.

有时候, 出现问题的地方并不是某个特定的函数, 而是文件中某一行的附近. 在这种情况下应该使用第二条命令, 它在文件的某一行插入断点.

如果开发人员已经跟踪到某个位置, 并且希望当下次执行时在这里停住, 这时候就可以用最后一个命令. 该命令在当前文件的当前行插入一个断点, 当然是在调试器内.

可以在任意时刻, 用下面的命令得到全部的断点:

```
:breaklist
```

当断点不再需要时就可以把它删除, 和添加断点一样, 有若干种删除断点的方法.

最简单的一种是用命令 `:breaklist` 找到断点的编号, 然后用下命令删除它:

```
:breakdel number
```

其他删除办法与添加断点的方法相同, 只不过要把 `breakadd` 改成 `breakdel`:

```
:breakdel func linenumber functionname
:breakdel file linenumber file
:breakdel here
```

如果想要一次性删除全部的断点, 执行:

```
:breakdel *
```

189

^①应该是个笔误 — 译者注



当在命令行启动 Vim 的调试模式时, 此时就可以插入一个断点, 方法是使用 `-c` 参数:

```
vim -D -c 'breakadd file 15 */.vimrc' somefile.txt
```

7.4 发布 Vim 脚本

脚本开发完成后, 就可以发布它们了 (当然, 也可以不这样做). Vim 在线社区已经成为实际上的脚本发布与搜索集散地, 因此, 建议读者也在这里发布自己开发的脚本. 但是在发布之前, 还需要准备一些东西.

首先, 开发人员需要判断自己的脚本是否需要打包成一个压缩文件, 比如 ZIP 文件, 又或者说是否就以单个的 .vim 文件进行发布. 选择第一个做法的理由是脚本包含了多个文件 (这些文件可能包含主要的脚本文件, 文件类型插件, 语法文件, 以及文档等等).

如何创建 ZIP 文件不在本书的讨论范围之内, 这里只介绍笔者是如何把待压缩的文件准备好的:

- 创建出的 ZIP 文件需要包含脚本文件所在的目录, 这里的目录指的是相对于 VIMHOME 的目录, 比如, 假设用户开发的脚本包含了:

```
VIMHOME/plugin/myscript.vim
VIMHOME/syntax/mylang.vim
VIMHOME/doc/myscript.txt
```

那么, ZIP 文件就应该包含这三个目录: plugin, syntax, 与 doc. 这种做法使得安装更加方便: 只要把 ZIP 文件解压到 VIMHOME 目录下即可.

- 总是为脚本配上一个帮助文件. 帮助文件应该安装到 VIMHOME/doc/, 帮助文件至少应该描述这个脚本是什么, 有哪些设置, 以及如何使用它.

即使脚本只由单个文件组成, 也最好把它和帮助文件一起打包成一个 ZIP 文件. 这可以提醒开发人员时刻更新文档. 关于如何创建 Vim 文档, 将在下一节进行更深入地讨论.

190

7.4.1 制作 Vimball

另一种发布方式是制作 Vimball 文件. 在前面已经介绍过如何用 Vimball 来安装脚本, 现在来看一下如何制作 Vimball 文件.

制作 Vimball 的命令是:

```
:[range]MkVimball filename.vba
```

看起来很简单, 不是吗? 不过在调用这个命令之前, 有一些准备工作需要完成.

第一件事是打开一个新的空白缓冲区, 用命令:

```
:enew
```

然后, 写上所有相关的文件路径 (相对于 VIMHOME 目录), 每行一个. 例如, 对于上面提到过的 ZIP 文件来说, 它包含的文件有:

```
plugin/myscript.vim
doc/myscript.txt
syntax/mylang.vim
```

填写完成后, 就可以开始执行 `:[range]MkVimball filename.vba`, 注意, 要把命令中的 `[range]` 替换成文件列表的起始与结束行号. 如果不想输入起始与结束行号, 则可以这样做: 把光标移动到文件的第一行, 切换到普通模式, 按下 `Shift+v`, 然后向下移动光标, 选中所有的行, 最后执行:

```
:MkVimball myscript.vba
```

Vim 自动地把选中的行的起始与结束行号添加到命令的前面. 命令中的文件 `myscript.vba` 可以是任意的名字, 如果文件已经存在, Vim 就会发出警告, 但不会覆盖文件.

如果开发人员确实想要覆盖掉原来的文件, 就在 `MkVimball` 后面加上 `!`. 命令执行结束后, 你就有了一个名为 `myscript.vba` 的 **Vimball** 文件, 别人可以用它来安装你的脚本.



在安装 **Vimball** 文件之前, 先要确保 Vim 已经安装了 **Vimball** 脚本. **Vimball** 脚本的最新版可以到 http://www.vim.org/scripts/script.php?script_id=1502 上下载.

191

7.5 注意文档

Vim 有一个非常完善的帮助系统, 几乎涵盖了 Vim 的方方面面. 用户安装完其他人开发的脚本后, 如果想要搜索相关的帮助信息, 此时会发生什么? 如果开发人员没有在脚本的安装包中添加文档, 那么用户就无法在 Vim 的帮助系统中找到相关的信息. 所以, 请在发布脚本时, 加上相应的文档.

现在来看一下如何创建具有链接, 标记等信息的 Vim 文档.

一个 Vim 文档文件就是一个普通的文本文件, 只不过带有一些特殊的标记. 当创建一个新的文档文件时, 开头第一行是最重要的, 先来看一个例子:

```
*docname.txt* single line of description
```

每一个文档文件的第一行的第一个字符必须是 `*`.

`docname.txt` 是当前正在编辑的文件名. 当 Vim 帮助系统需要链接到本地附加文件 (参考 `:help local-additions`) 时就会用到这个信息. 第二个 `*` 后面的内容是关于本文档的简短描述. 对于前面开发的脚本来说, 可以把文档的第一行写成:

```
*myscript.txt* Documentation for example script myscript.vim
```

写完这一行后, 就可以开始编写文档的实际内容.

比较典型的做法是以一长串的介绍开始, 详细介绍本文档的主题内容. 这些内容可以包括作者的名字与联系方式. 然后, 如果文档比较长的话, 最好添加一个目录 (前面给的例子就包含了一个目录). 对于 `myscript.vim` 来说, 它的文档开头可以这样写:

```
*myscript.txt* Documentation for example script myscript.vim
```

```
Script   : myscript.vim - Example script for vim developers
Author   : Kim Schulz
```

```

Email      : <kim@schulz.dk>
Changed    : 01/01/2007
=====

                                *myscript-intro*

1. Overview~

This document gives a short introduction to the example
script myscript.vim.
This script is made as an example for vim users on how to
structure a simple vim plugin script such that it is easy
to read and figure out.
The following is covered in this document:

1. Overview          |myscript-intro|
2. Mappings          |myscript-mappings|
3. Functions         |myscript-functions|
4. Todo              |myscript-todo|
=====

```

192

这个例子用到了 Vim 文档中大部分会用到的格式化标记。现在一个个地加以介绍。

第一个标记是 `*...*`，它把关键词标记成 Vim 帮助系统可以跳转到的位置。在这里我们把它写成了 `*myscript-intro*`，这样的话，就可以用下面的命令跳转到该文档：

```
:help 'myscript-intro'
```

下一个标记是 Overview 后面的 `~`，这个标记可以使 Overview 带上不同的颜色。

然后是目录中某些关键词两边的 `|`。这个标记创建了一个指向特定章节的链接，章节的名称用 `*...*` 标记。由多个等号组成的一行指出了章节的边界，但它们并不是真正的标记，没有特殊的功能。

文档中接下来的各小节用同样的格式编写，但是，如果它们包含了一小段 Vim 代码，那就需要另一个标记来指明，比如包含函数的章节就有可能含有 Vim 代码，先来看一个具体的例子：

```

=====

                                *myscript-functions*

3. Functions~

Besides the functions available via mappings (as described
in |myscript-mappings|) there are some extra global func-
tions available.

MyglobalfunctionB()~

This function is one of the global functions in this script.
An example of usage could be: >

    :call MyglobalfunctionB()

<

```


Vim returns:

Hello from the global-scope function myglobalfunctionB~

MyglobalfunctionC()~

This function is a global function that also calls one of the internal functions ("s:MyfunctionA()") in the script.

An example of usage could be: >

:call MyglobalfunctionC()

<

Vim returns

Hello from MyglobalfunctionC() now calling locally:~

This is the script-scope function MyfunctionA speaking~

=====

需要注意的标记是代码周围的 >...<, 它用来标记代码, 另外, 还用 ~ 标记了函数的返回信息.

为了创建一份良好的文档, 读者所需要知道的就是这些了.

当用户想要安装文档时, 首先得把文档放到 VIMHOME/doc/ 目录下, 然后在 Vim 中执行:

```
:helptags docdir
```

命令中的 docdir 是相对于 VIMHOME/doc/ 的路径. 如果文档中的某个关键词已经被其他人使用过了, Vim 就会发出警告, 此时开发人员应该修改关键词, 然后重新发布文档.



想让自己的文档以多种语言发布? 请参考 :help 'help-translated'.

7.6 使用外部解释器

虽然开发人员几乎可以用 Vim 脚本完成任意的工作, 不过有时候借用其他语言可能会更好一点. Vim 开发人员很早以前就已经注意到这点, 因此他们在 Vim 中加入了对其他脚本语言的支持. 在这些脚本语言中, 读者要重点关注以下三种:

- Perl
- Python
- Ruby

接下来的小节将会简单介绍如何在 Vim 脚本中使用这三种语言.

Vim 默认上不支持这三种语言, 为了解决这个问题, 读者既可以选择自己重新编译 Vim (在编译前开启相应的编译选项), 或者安装一个已经支持它们的 Vim 版本.

为了查看系统中的 Vim 是否支持这三种脚本语言, 在 Shell 中执行:

```
vim --version
```

注意命令的输出是否包括以下内容:

```
+perl
+python
+ruby
```

名字左边的 + 表示 Vim 支持该语言, 如果是 -, 比如 -perl, 则表示该版本 Vim 不支持 Perl.

除了在命令行查看, 还可以在启动 Vim 后, 用函数 has() 来检查:

```
:echo has("perl")
:echo has("python")
:echo has("ruby")
```

如果支持, 则命令返回 1.

195

7.6.1 Perl

Perl 是一门非常流行的脚本语言, 在文本处理方面非常强大, 在 Vim 中同样有用.

在 Vim 使用 Perl 的最简单方式是:

```
:perl command
```

这表示执行 Perl 命令 command. 注意, 在 Perl 命令中设置的值在整个 Vim 会话期间都一直有效.

用户经常会一次执行多个 Perl 命令, 这时候可以用下面这种形式的执行方式:

```
:perl << endpattern
perl code here
endpattern
```

上面的命令会执行第一个 endpattern 与最后一个 endpattern 之间的所有 Perl 代码.



对于 Perl, Python, 或 Ruby, 用户可以用任意的字符串作为 endpattern, 但是在最后一行只能出现该字符串, 而且它还必须出现在一行的开始. 如果省略了第一个 endpattern, 则 Vim 默认把句点当作 endpattern.

往 Vim 的编辑窗口中打印一行的代码可以这样写:

```
:perl << EOF
VIM::Msg("this is a text");
EOF
```

现在, EOF 是 endpattern, 在 Perl 代码中, 函数 VIM::Msg() 向 Vim 打印一条消息. 除了这个函数, 还有许多函数可以用来沟通 Vim 与 Perl, 比较常见的有:

- VIM::buffers(): 返回所有打开的缓冲区列表.
- VIM::SetOption("option"): 在 Perl 中设置 Vim 的 option 选项.
- \$curbuf->Name(): 返回当前缓冲区的名字.
- \$curbuf->Set(100, "fooo"): 把当前缓冲区的第 100 行文本设置成 fofoo.

- `$curwin->SetCursor(15,8)`: 把光标的位置移动到当前窗口的第 15 行, 第 8 列

196

可以用下面的命令获取可以在 Perl 中执行的所有 Vim 特定函数:

```
:help perl-pview
```

如果开发人员决定在脚本在加上 Perl 代码, 要记得检查用户的 Vim 版本是否支持 Perl.

始终把 Perl 代码封装在 Vim 函数中是个不错的主意. 这种方法很容易实现, 而且对于经验不主的用户来说, 脚本看起来与其他 Vim 脚本没什么区别. 把 Perl 代码封装在 Vim 函数中的例子可以这样写:

```
function MoveCursor(row,col)
    if has("perl")
        perl << EOF
        ($oldrow,$oldcol) = $curwin->Cursor();
        VIM::Msg("Old position was: ($oldrow,$oldcol)");
        $curwin->Cursor(row,col);
    EOF
    else
        echo "perl not available. canceling function call"
    endif
endfunction
```

上面的函数首先获取当前窗口的光标位置, 并打印它们, 然后把光标移动到由参数指定的位置上.

如果当前的 Vim 版本不支持 Perl, 函数就会打印一条关于这个事件的消息. 注意, 即使其他代码是缩进过了的, 但我们仍然把 EOF 写在一行中最靠左的位置上. 为了能让 Vim 准确识别出 *endpattern*, 用户必须严格遵守这条规则.

7.6.2 Python

在最近的这几年中, Python 已经成为众多程序员最喜欢的脚本语言之一. 这主要是因为它的易用性, 以及对缩进的严格规定 (缩进提高了代码的可读性).

和 Perl 一样, Python 也有一些接口同 Vim 交互. 在 Vim 中使用 Python 主要有以下三种方式:

197

1. 如果只是想在 Vim 中执行一条 Python 语句, 可以用 `:python statement`, 例如:

```
:python print "hello Vim developer"
```

2. 如果想要一次执行大量的 Python 代码, 可以用:

```
:python << endpattern
python statements here
endpattern
```

它们执行 *endpattern* 之间的所有 Python 代码.

3. 最后一种方法是在 Vim 中执行一个 Python 脚本, 比如:

```
:pyfile file.py
```

可以把 `file.py` 替换成任意一个你想要执行的 Python 脚本。

有时候, Python 脚本可能需要从命令行获取一些参数, 如果使用的是 `:pyfile`, 就做不到这点。不过, 可以通过设置 `sys.argv` 解决这个问题, 一个使用示例是:

```
:python import sys
:python sys.argv = ["argument1", "argument2"]
:pyfile myscript.py
```

为了更方便地与 Vim 交互, Python 包含了一个称为 `vim` 的模块, 这个模块可以让 Python 脚本访问 Vim 的许多额外功能, 一个使用示例是:

```
import vim
window = vim.current.window
window.height = 200
window.width = 10
window.cursor = (1,1)
```

下面的命令可以用来获取所有的可用函数列表:

```
:help 'python-vim'
```



如果要在 Vim 脚本中使用 Python 代码, 最好把它们封装在 Vim 函数内。

7.6.3 Ruby

在西方国家, Ruby 直到最近才成为一门流行的编程语言, 但是早在这之前, Ruby 就在亚洲流行开了, 而且, 自从它成为 Web 开发的脚本语言之后, 越来越多的程序员开始喜欢上它。Ruby 最大的优点是它是一门真正的面向对象编程语言, 因此 Ruby 是模块化的。

Vim 对 Ruby 的支持非常完善, 因此用户可以在 Vim 中运行 Ruby 代码, 执行的方式有很多种, 其中最简单的一种是 `:ruby command`。

把 `command` 替换成任意一个单行的 Ruby 命令, 比如:

```
:ruby print "Hello from Ruby"
```

如果用户想要同时执行多行 Ruby 代码, 可以这样写:

```
:ruby << endpattern
python commands here
endpattern
```

上面的命令会执行两行 `endpattern` 之间所有的 Ruby 代码。如果在 Ruby 代码中设置了某个变量, 或创建了某个对象, 那么在整个 Vim 会话期间, 它们都是可用的。

一个使用示例是:

```
:ruby << EOF
  window = VIM::Window.current
  window.height = 250
  window.width = 35
  window.cursor = (10,10)
EOF
```

如果用户想要执行的 Ruby 代码存放在某个文件中, 可以用下面的命令执行文件中的代码:

```
:rubyfile file.rb
```

上面的命令还可以写成:

```
:ruby load 'file.rb'
```

在 file.rb 中创建的所有对象与变量在整个 Vim 会话期间都是可用的, 除非显式删除了它们。

199

为了与 Vim 交互, Ruby 提供了一个专门用于 Vim 的模块, 叫做 vim, 这个模块包含了许多访问与设置 Vim 的方法, 其中包括:

- `VIM::Set_option('option')`: 设置 Vim 的 *option* 选项
- `VIM::Message("message")`: 在 Vim 中打印一条消息
- `$curwin.height`: 当前窗口的高度
- `$curbuf.width`: 当前缓冲区的宽度
- `VIM::Buffer.current.append(10, "line")`: 在当前缓冲区的第 10 行新增一行文本
- `VIM::Buffer.current.length`: 返回当前缓冲区的行数

用下面的命令获取所有的可用方法与对象:

```
:help 'ruby-vim'
```



在使用 Ruby 代码之间, 要记得始终检查用户的 Vim 版本是否支持 Ruby.

7.7 小结

这一章主要是为那些想开发 Vim 脚本的人写的.

在介绍了 Vim 脚本的各个组成要素后, 现在要把它们都组装到一个完整的 Vim 脚本中, 我们采用了自顶向下的方法, 并且逐行地介绍了脚本中的每一行.

一个很重要的一点是, 脚本开发人员无法预知用户是如何配置它们的 Vim 的, 因此, 脚本需要检查某个设置是否可用, 而且不能破坏用户已有的设置.

我们学习了如何在脚本作用域内使用变量与函数, 这样就可以避免全局作用域受污染. 通过限定作用域, 可以确保用户只能访问到和他们相关的功能.

在查看了几个 Vim 脚本的结构之后,介绍了如何检查一些先决条件,比如操作系统与 Vim 版本,以便在继续执行之前,判断某些条件是否满足。

200

学习了如何创建 Vim 脚本之后,介绍了如何调试 Vim 脚本.这一节介绍了 Vim 的调试模式,以及如何逐行地执行代码。

脚本编写完成并且没有错误后,接下来就可以发布脚本了.这一节介绍了如何发布脚本,这样的话,其他人就可以安装并使用你的脚本。

文档是已发布脚本非常重要的一部分,这一节介绍了如何使用 Vim 标记来编写文档.通过使用这些标记,Vim 的帮助系统就可以检索到开发人员自己编写的文档,而且在文档中跳转也很方便。

对于有些人来说,光使用 Vim 脚本语言还不足以满足他们的需求,因此接下来介绍如何在 Vim 中使用其他脚本语言。

我们介绍了 Vim 中最流行的三种脚本语言 — Perl, Python, 和 Ruby. Vim 默认不支持这三种语言,所以在编译时必须手动开启相应的编译选项,除了重新编译,还可以直接从网络上下载支持它们的已编译安装包。

我们逐一介绍了如何在 Vim 中执行这三种脚本语言,对于 Python 与 Ruby, Vim 还可以直接执行整个 Python 或 Ruby 脚本文件。

我们还看到这三种语言都提供了用于和 Vim 交互的模块,这些模块可以访问 Vim 的许多信息,因此开发人员可以直接在这些语言脚本中控制 Vim。

看到这里,读者就有能力自己编写 Vim 脚本,调试它们并发布.附录将会看到某些 Vim 用户开发的脚本几乎可以让 Vim 无所不能。

附录 A 无所不能的 Vim

201

据说 Vim 可以做任何事情, 虽然这可能不是真的, 但 Vim 的确可以完成许多读者根本想都不会想到的事情. 这一节介绍一些原来需要通过 Vim 脚本, 或外部工具才能完成的的事情. 这一节讨论的内容涵盖了游戏, 邮件客户端, IRC 即时聊天, 与集成开发环境的设置.

A.1 Vim 游戏

虽然 Vim 只是一个文本编辑器, 但仍然有很多人开发了大量的脚本, 来让 Vim 完成其他一些非编辑任务, 其中包括可以直接在 Vim 中玩的小游戏. 这些小游戏不是那些类似于“20 个小问题”那样的文本类游戏, 而是带有图形界面的. 这些图形界面并不完美, 因此它们是用 ASCII 字符组成的界面, 但对于游戏来说已经足够了.

202

A.1.1 生命游戏

第一个要介绍的游戏严格说来, 并不能算作游戏, 但是仍然值得说一下. 生命游戏 (The Game of Life) 通常被人称为无玩家游戏, 因为这个游戏并不需要玩家参与, 只需要静静地看着就行. 游戏是一个非常简单的人工智能, 用来模拟细胞的演化. 细胞要遵守下面几条规则:

1. 在一个活的细胞周围, 如果活细胞数少于两个, 则这个细胞会由于孤独而死去
2. 在一个活的细胞周围, 如果活细胞数多于三个, 则这个细胞会由于拥挤而死去
3. 在一个活的细胞周围, 如果活细胞数是两个或三个, 则这个细胞会继续存活下去
4. 在一个死的细胞周围, 如果活细胞数是三个, 则这个细胞会复活

1996 年, 一个自称为大胡子艾丽 (Eli the Bearded) 的人开发了一个 Vim 脚本, 该脚本实现了生命游戏. 脚本的运行速度并不是非常快, 仅仅是为了阐明游戏的原理. 对于大多数人来说, 这个游戏非常的枯燥, 但是对于生命游戏迷来说, 这是个非常有趣的实现.

可以到下面这个网址下载生命游戏的 Vim 脚本: <http://www.vanhemert.co.uk/vim/vimacros/life1.vim>.

A.1.2 贪吃蛇

1986 年, 我拥有了第一台 PC, 这台 PC 上只有一个游戏可以玩 — 贪吃蛇. 我花了很长时间玩这个游戏, 游戏的内容是控制一条小蛇的爬行, 有很多个关卡, 在每个关卡中, 小蛇都要吃完一定数量的方块后才能过关, 随着方块的增多, 小蛇身体的长度也在加长. 规则是小蛇不能穿越边界, 也不能穿越自己的身体.

2004 年, Hari Krishna Dara 重新用 Vim 脚本实现了这个游戏. 虽然他的游戏只包含很少的关卡, 但是只要用户需要, 就可以添加关卡. 虽然游戏在运行时需要不停地打印文本, 但是看起来非常得流畅.

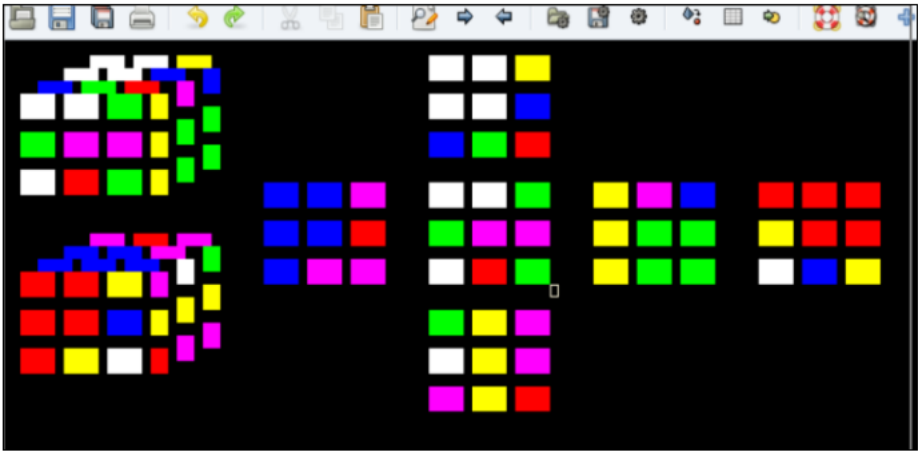


脚本的下载地址是 http://www.vim.org/scripts/script.php?script_id=916.

A.1.3 魔方

1974 年, 匈牙利雕刻家兼建筑学教授 Erno Rubik 发明了一个由 27 个小方块组成的魔方. 魔方的每一面分别涂上了一种不同的颜色. 如果试图旋转魔方, 那么魔方的颜色就会变乱, 游戏的目标就是把一个弄乱了的魔方还原成最初的样子.

在 Rubik 发明了魔方的 30 年后, 2005 年, Olivier Vermersch 用 Vim 脚本开发了一个魔方游戏. Vim 版的魔方同样会让玩家花费几个小时的时间, 把魔方复原.

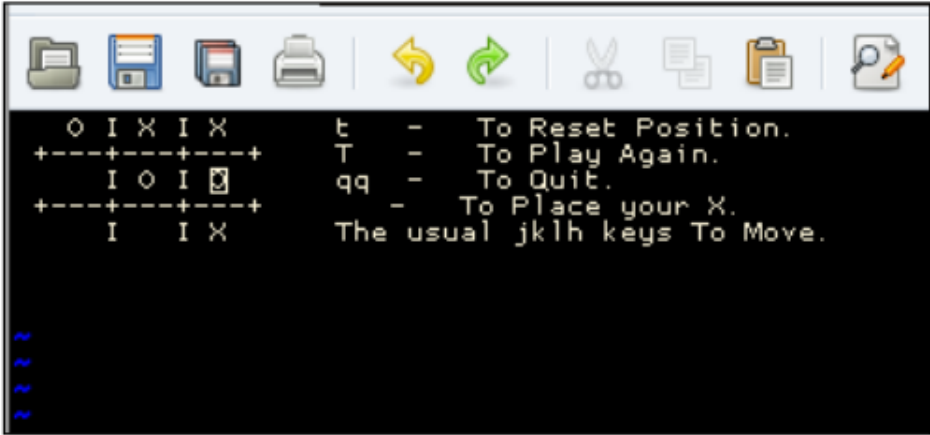


游戏的使用方法与脚本的下载地址是 http://www.vim.org/scripts/script.php?script_id=1271

A.1.4 井字棋

应该有很多小孩玩过井字棋 (Tic-Tac-Toe) 这个游戏.

1996 年, Kevin Earls 用一系列宏命令在 Vim 中实现了这个游戏, 游戏双方分别是人类玩家和 Vim.



虽然 Vim 的智能不是很先进,但是想要赢它还是比较困难的。
可以到 Earls 先生的个人主页上下载到游戏脚本,脚本中包含了游戏的安装与使用方法,下载地址是 <http://www.vanhemert.co.uk/vim/vimacros/ttt.vim>.

A.1.5 扫雷

1995 年,微软在新发布的操作系统 Windows 95 中,预装了一个称为扫雷的小游戏.这款游戏混合了数字谜语和一点运气.玩家的目的是找出雷区中所有的地雷:或者是点击没有地雷的方块,或者是在有地雷的方块上插一把旗子.方块下的数字表示周围 8 个小方块下的地雷个数.如果玩家计算出了地雷所在的位置,就在地雷所在的方块上插一把小旗子.

205

2004 年,Charles E. Campbell 用 Vim 脚本重新实现了这个游戏,游戏包含了多个难度等级,脚本的下载地址与使用方法在 http://vim.sourceforge.net/scripts/script.php?script_id=551.

A.1.6 推箱子

我比较喜欢智力游戏,也很喜欢推箱子.游戏玩起来非常简单,但是游戏本身却可以非常得难.游戏的内容是让一个小人把箱子推到指定的位置上.听起来很简单是吗?游戏的规则是一次只能推一个箱子,而且只能推,不能拉,因此,如果把箱子推到角落中,那就再也推不动它了,这时候玩家就不得不重新开始游戏.

2002 年, Mike Sharpe 用 Vim 脚本实现了这个游戏,游戏的关卡设计来自以前的 Linux 游戏 XSokoban. Mike Sharpe 尽量保持了用户接口的简单,但游戏玩起来仍然十分有趣.



XSokoban 游戏可以到这个网址下载: <http://www.cs.cornell.edu/andru/xsokoban.html>.

游戏脚本的下载地址是 http://www.vim.org/scripts/script.php?script_id=211.

206

A.1.7 俄罗斯方块

最后要介绍的一款游戏是真正的经典 — 俄罗斯方块,游戏中会有不同大小与形状的方块落下来,玩家的目的是适当地对它们进行排列,尽量使得每一行都是满的.俄罗斯方块是前苏联人 Alexey Pajitnov 在 1985 年发明的,在这之后,几乎每个平台都实现了这个游戏,而且产生了数以百计的变种.



2002 年, Gergely Kontra 用 Vim 脚本实现了这个游戏, 游戏拥有不同的模式, 还可以记录以前得过的最高分. 脚本的下载地址与使用方法在 http://www.vim.org/scripts/script.php?script_id=172.

A.2 集成开发环境

在书中我几次提到, Vim 可以帮助程序员完成很多事情, 从我个人来说, 几乎所有的编程工作都是用 Vim 完成, 但有时候会听到其他程序员说: “这么原始的编辑器你是怎么用的?”, “没有了集成开发环境, 你是怎么编程的?”. 其实, Vim 同样可以提供集成开发环境.

首先来看一下集成开发环境应该提供哪些功能.

207

一个典型的集成开发环境, 比如 Microsoft Windows Visual Studio®, 包括了下面这些组件:

- 支持自动缩进, 语法高亮, 自动补全的编辑器
- 集成的编译器, 如果编译出错, 可以直接跳转到出错的地方
- 集成的调试器
- 文件浏览器
- 项目浏览器
- 标签浏览器, 通过它可以快速查看变量, 函数, 方法, 类
- 支持在文件间, 或定义间快速跳转
- 可能还集成了版本控制系统

那么, Vim 是否支持上面所说的全部特性呢? 我们逐条分析各个特性, 看一下 Vim 如何才能支持它们.

第一条是显而易见的, Vim 本身就是一个功能丰富的编辑器.

第二条是编译器的集成. Vim 通常用于编程工作, 因此它本身就支持编译器集成. 对常见的编程语言来说, 和编译器集成相关的设置在 Vim 中本来就是设置好了的. 如果不是, 可以用下面的方法查看如何设置它们:

```
:help compiler
```

编译器集成通常会用到另一个功能: quickfix 列表. 当出现编译错误时, 程序员可以通过这个列表跳转到出错的代码, 在各个错误之间前进或后退, 修正错误, 然后重新编译.

关于 quickfix 列表的更多信息, 可以查看:

```
:help quickfix
```

下一条是集成的调试器. 和前一条不同的是, 对调试器的支持并不是 Vim 的标准特性. 但是有几个脚本可以帮助 Vim 集成调试器. 如果是 Linux, 可以用 VimDebug, 这个脚本可以让 Vim 集成 gdb (C/C++ 调试器), jdb (Java 调试器), pdb (Python 调试器), 以及 Perl 调试器. 脚本的下载地址是 http://www.vim.org/scripts/script.php?script_id=663.

208

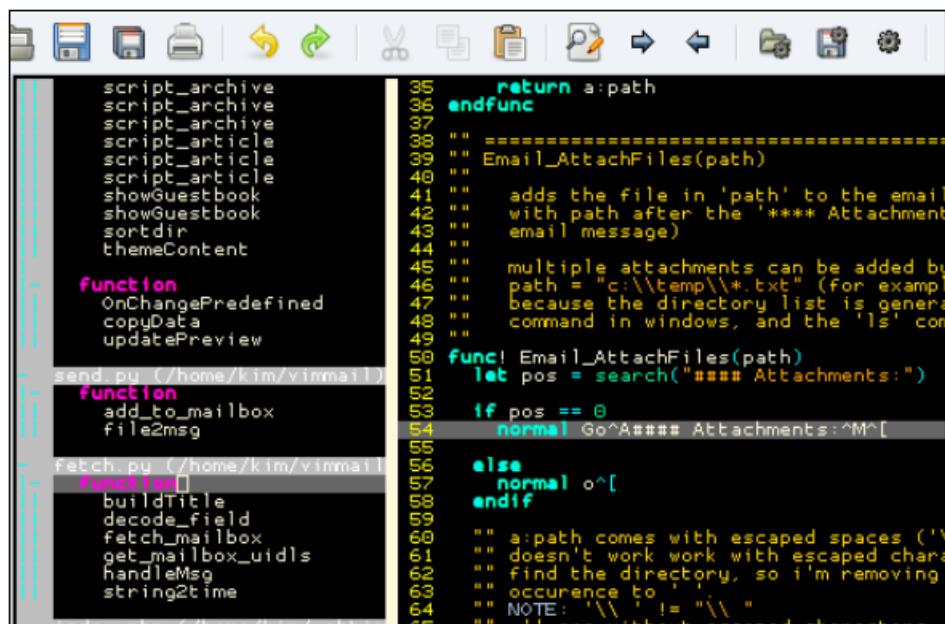
另外一个功能更加丰富的调试器集成是 Clewn, 它集成了 gdb 调试器与 Vim, 支持调试器的所有功能, 它甚至还支持远程调试. Clewn 的下载地址是 <http://clewn.sourceforge.net>.

再下一条是文件浏览器. Vim 本身就带有一个文件浏览器, 不过有一个脚本可以让文件浏览器与集成开发环境更加搭配, 这个脚本是 VtreeExplorer. VtreeExplorer 不仅可以浏览文件, 还可以把目录按照树状结构显示出来, 树的结点可以打开或折叠. 脚本的下载地址是 http://www.vim.org/scripts/script.php?script_id=184.

接下来是集成开发环境的项目管理部分. 项目浏览器必须支持文件与项目的关联, 这样的话, 当用户打开某个项目时, 项目内的所有文件就可以同时被打开. 如果用户用的是 Gvim, 就可以用 ProjMgr 脚本. ProjMgr 创建一个菜单项, 该菜单项包含了一张可用项目列表, 除此之外, 它还可以创建新项目. 脚本的下载地址是 http://vim.sourceforge.net/scripts/script.php?script_id=279, 控制台版本的下载地址是 http://www.vim.org/scripts/script.php?script_id=69.

对于标签浏览器, 有两种解决办法, 其中之一是执行标签名, 函数名, 和其他名字的补全命令, 然后逐一浏览, 另一种办法是将所有可用的标签列在一个单独的窗口中, 为了创建这样一个窗口, 我推荐 TagList 脚本. 只要是 Ctags 程序支持的编程语言, TagList 都支持. 脚本会创建一个窗口, 这个窗口列出了所有的定义, 函数, 方法, 和类, 不仅浏览方便, 还可以快速跳转到声明标签的地方. TagList 的下载地址是 <http://vim-taglist.sourceforge.net>.

209



Vim 本身提供了两个命令: `gf` 与 `gd`, 分别用来跳转到标签所在的文件与声明标签的地方, 通过这两个命令的帮助, 用户可以在文件中快速地移动。

最后要谈到的是版本控制系统, 比如 CVS, SVN 和 Perforce. 版本控制系统的集成同样可以通过脚本来实现. 对于前面提到的三个版本控制系统, 我推荐以下几个脚本:

- CVS 与 SVN 相关的脚本在 http://www.vim.org/scripts/script.php?script_id=90
- Perforce 相关的脚本在 http://vim.sourceforge.net/scripts/script.php?script_id=240

用来组装集成开发环境的各个组件都已经准备好了, 现在所要做的就是将它们都集成起来. 可能有用户怕麻烦, 不想自己动手完成最后一步, 不用担心, 已经有其他人帮你做好了. 比如 Vim JDE (Just a Development Environment), 它集成了上面提到的所有脚本, 对 Java, C/C++ 程序员来说, 这是一个功能丰富的集成开发环境. Vim JDE 的下载地址是 http://www.vim.org/scripts/script.php?script_id=1213. 另一种办法是在 Vim 社区中搜索集成脚本的方法, 然后按照方法所说的步骤执行, 比如 http://vim.wikia.com/wiki/Using_vim_as_an_IDE_all_in_one.

210

A.3 邮件程序

一直以来, Vim 就支持与多种不同的程序进行集成, 其中就包括一些邮件客户端, 比较有名的客户端是 Mutt.

对于某些 Vim 用户来说, 与邮件客户端集成还远远不够, 于是他们决定用 Vim 脚本实现一个完整的邮件客户端, 客户端的功能包括在 Vim 中直接收发和管理邮件.



开源邮件客户端 Mutt 的下载地址是 <http://www.mutt.org/>.

2004 年, Suresh Govindachar 为 Vim 开发了 The Mail Suite (TMS), 这个软件部分用 Vim 脚本实现, 还有一部分用 Perl 脚本 (嵌入在 Vim 脚本代码) 实现. TMS 支持邮件客户端的全部功能, 同时还可以用到 Vim 提供的所有特性.

TMS 的下载地址是 http://www.vim.org/scripts/script.php?script_id=1052.

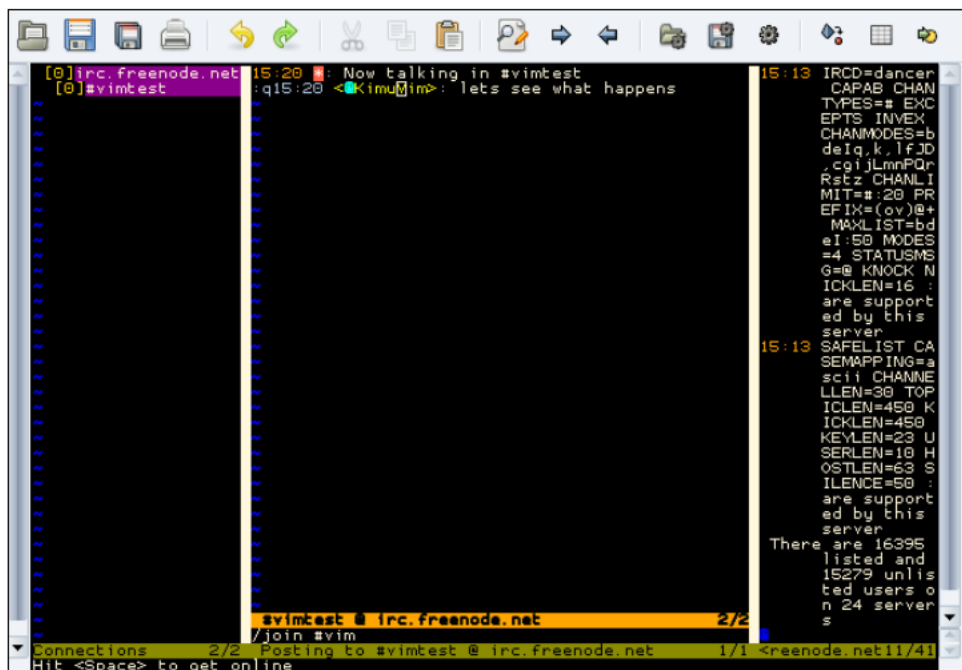
2005 年, David Elentok 用一个 Vim 脚本和两个 Python 脚本实现了邮件客户端. 虽然目前脚本还在开发阶段, 但邮件客户端的主要功能已经完成.

Elentok 的脚本的下载地址是 http://www.vim.org/scripts/script.php?script_id=1346.

211

A.4 聊天

在如今这个时代, 几乎找不到一台没有安装聊天程序的计算机, 但在过去可不是这样. 回到 20 世纪 80 年代后期, 当时因特网上的聊天仅限于 BBS 的公告板. 到了 1988 年, 芬兰人 Jarkko Oikarinen 开发一个客户端/服务器模式的聊天程序 — Internet Relay Chat (IRC). 在接下来的十年, IRC 成为了一款非常流行的聊天程序, 越来越多的人开始加入到聊天网络中. IRC 的使用方法非常简单: 下载一个 IRC 客户端, 用客户端连接到 IRC 服务器, 加入到一个你最喜欢的聊天频道, 接下来就可以和别人畅所欲言了.



IRC 客户端版本众多, 2004 年, 日本人 Madoka Machitani 在 Vim 中实现了一个 IRC 客户端, 他把它叫作 VimIRC. VimIRC 功能丰富, 支持同时在多个网络的聊天频道中聊天, 在客户端中还可以使用 Vim 命令, 比如 I, /, ? 等.

212

不过, 在 Vim 中聊天会有一个小问题. 为了让聊天保持在线, 客户端会执行一个循环来不停地更新所有的东西. 如果用户移动到另一个和 IRC 客户端无关的缓冲区, 循环就会停止, 用户的聊天网络最终就会断开. 解决这个问题的技巧是让客户端在自己的 Gvim 或 Vim 中启动, 这可以在命令行中完成:

```
gvim -i NONE -i .vimircrc -c VimIRC
```

命令行中的 .vimircrc 是 VimIRC 配置文件的路径.

VimIRC 的下载地址是 http://www.vim.org/scripts/script.php?script_id=931.



Vim 用户常去的 IRC 网络与频道分别是 `irc.freenode.net` 和 `#vim`.

A.5 Twitter 客户端

最近几年社交网络变得越来越流行, Twitter (<http://www.twitter.com>) 是最流行的社区平台之一, 它的主要功能是和朋友分享“状态更新”.

Po Shan Cheah 开发了一个 Vim 脚本, 使用这个脚本就可以在 Vim 中阅读朋友的状态更新, 还可以直接发送自己的状态更新, 以及向 Twitter 朋友发送私人消息.

脚本的名字是 TwitVim, 下载地址是 http://www.vim.org/scripts/script.php?script_id=2204.

脚本的输出非常简单, 主要是列出用户所请求的朋友的最近一次状态更新, 它的示例输出如下图所示:

213

```
User timeline*
*****
kimschulz: @digsby: whats up with this error when posting to FB: Error posting comment.^@4: A
plication request limit reached |01:19 Jan 07, 2010|
kimschulz: Psystar Switches to Linux, Temporarily Halts Sales of Rebel EFI http://bit.ly/409v
Z ( |10:45 Dec 29, 2009|
kimschulz: Så fik jeg også tid til at prøve @Dropbox. Ganske smart til fildeling og sync med
omputere. 2GB gratis plads! http://tinyurl.com/yjob8vc |11:31 Dec 22, 2009|
kimschulz: just installed handwriting hotfix for my @htc diamond2 but no handwriting is avail
ble....oh well may it does not work on danish wm6.5rom |11:12 Dec 18, 2009|
kimschulz: RT @pocketnowTweets: Nexus One Makes a Video Debut http://tinyurl.com/ydack9h |12:
2 Dec 16, 2009|
```

TwitVim 使用命令行程序 cURL (<http://curl.haxx.se/>) 从 Twitter 上抓取信息, 因此在使用 TwitVim 之前, 要先下载安装 cURL.



关于 Twitter 的更多信息, 请登录 <http://www.twitter.com>, 在那里可以免费注册一个账号.

附录 B Vim 配置管理

215

第二章介绍了 Vim 的主要配置文件,随着阅读的深入,我们不断地往 `vimrc` 中添加新的配置信息,最终,配置文件可能会变得非常混乱,难以管理.

本章将介绍一些组织 `vimrc` 的方法,从小技巧一直到整个配置系统.

最后,将会介绍如何在多个不同的计算机中使用同一个 `vimrc` 文件,方法是在网络中保存一份副本.

B.1 保持 `vimrc` 整洁的技巧

`vimrc` 是用户设置 Vim 的核心文件,如果没有它,就只能使用系统中原有的设置,因此用户要时刻保持 `vimrc` 的整洁,并及时更新,只有这样,用户才能时刻知道文件中包含了哪些内容. 有时候,用户可能没办法在文件中找到自己想要的內容,笔者就曾经遇到过这种情况,当时我的 `vimrc` 超过了 2000 行,到那时才意识到整洁的必要性. 保持 `vimrc` 整洁并组织良好的技巧有:

216

1. 保持 Vim 处于非兼容模式

这个技巧可能并不会让 `vimrc` 更整洁,至少不能马上看出来,然而它却非常重要. 让 Vim 处于非兼容模式下就可以打开许多特性,而这些特性会被很多技巧和脚本使用到. 所以,最好在 `vimrc` 的第一行总是写上 `set nocompatible`.

2. 使用注释

有时候,用户会在 Vim 中改变一些设置,然后再把设置添加到 `vimrc`,过一段时间后,当用户想要清理 `vimrc` 时,就会发现自己已经记不住某段脚本是干嘛用的,以及当初为什么要添加它,甚至连代码是从哪儿来的都已经记不清了. 为了防止这种情况发生,那就给新加的东西写上注释. 笔者建议在注释中包含这些内容: 代码的作用 (描述),从哪儿得到的 (来源),代码的原作者. 有了注释的帮助,用户就能够方便地查询到代码的来源,以及决定是否需要删除它们. 注释用引号开始,比如 `"This is a comment`.

3. 数据分组

脚本通常需要一些额外的设置,或者是用户想要为脚本中的某些功能设置一些额外的按键绑定 (映射). 为了能更方便地看出设置的归属,最好为数据分组. 分组的条件有很多种,笔者推荐以下这些 (按照在文件中出现的顺序,从上到下排列):

- 通用的全局设置
- 自己私有的按键映射
- 特定于脚本的设置,按脚本分组
- 其他设置

4. 使用多个文件

有时候, `vimrc` 可能会变得非常巨大, 无论怎么组织都显得非常混乱. 对于这种情况, 最好把它切分成多个文件. 为了切分 `vimrc`, 首先把需要分出的内容剪切到另一个文件中, 文件名最好能描述出文件的作用, 然后在 `vimrc` 中原来的位置用 `source` 命令从新文件读取命令. 比如, 把和按键映射相关的设置命令都移动到 `mappings.vim`, 然后在 `vimrc` 中添加 `source $HOME/.vim/mappings.vim`.

217

5. 测试时使用其他文件

从前, 当笔者想要测试某段新的 Vim 脚本代码或者宏时, 通常会把它们添加到 `vimrc` 的末尾, 在测试结束后, 笔者往往会忘记把它们删除, 经过一段时间后, `vimrc` 就充满了我的测试代码. 为了避免重蹈覆辙, 读者应该把测试代码写在另一个文件中, 然后用 `source` 命令执行文件中的测试代码. 如果觉得不错, 就把代码移动到适当的地方, 否则, 只要把文件删掉即可.

6. 为不同的操作系统使用不同的文件

如果读者也像我一样, 在多个不同的系统中使用 Vim, 可能会发现有些配置不能适应全部的系统. 原因可能是因为有些设置是和操作系统相关的, 又或者是硬件配置不一样, 比如说有些系统的屏幕比较小, 而另一些则比较大. 如果出现这样的情况, 最好把系统相关的设置写在不同的文件中, 然后把通用的文件拷到所有的系统中, 再把系统相关的配置文件拷到各自的系统中.

如果读者能够严格遵守这些技巧, 就很有希望把 `vimrc` 保持在一个非常整洁的状态下, 这样的话, 无论想修改什么内容, 你都能快速地找到.

B.2 vimrc 配置系统

使用 Vim 时, 为了永久地更改某项配置, 用户需要打开配置文件, 修改后再保存. 如果只是在 Vim 中修改, 而没有写到配置文件中, 程序退出后, 所有的修改都会丢失.

假如有这样一个脚本, 它提供了一个设置菜单, 用户在其中所做的修改在程序退出后仍然有效, 那将会怎样? Jos van Riswick 就开发出了这样的脚本.

通过使用他的脚本, 并按照他的语法来创建 `vimrc`, 用户不仅可以使新的配置持久化, 还可以根据向导的指引, 一步步地对配置加以修改.

脚本的实现方式是, 把配置项周围的注释作为设置向导中所显示的信息的占位符. 笔者发现这个脚本非常聪明, 而且和我们通常用的不太一样, 所以我们在这里简单地介绍一下如何使用它.

218

脚本的下载地址是 http://www.vim.org/scripts/script.php?script_id=1894, 下载的压缩包中包含了下面这些文件:

- `setup.vim`: 主要的脚本
- `array.vim`: 包含了处理字符串数组的函数
- `arrayg.vim`: 包含了处理全局字符串数组的函数
- `strfun.vim`: 包含了字符串处理函数
- `tableaf.vim`: 用来在配置系统中显示 `tab leaf`

安装脚本的方法是把压缩包解压到 VIMHOME 目录下, 解压后, 文件就自动放到了 plugin/ 子目录中. 现在, 用户就可以准备好修改 vimrc, 但是在修改之前最好做个备份.

在我们开始往 vimrc 添加设置之前, 先来看一下配置系统的使用语法. 前面已经说过, 技巧就是配置项周围的注释. 这些注释包含了很特殊的语法, 这样脚本才能知道这些注释是干嘛用的, 语法的基本规则是 `"|ID|tabname|text|command|extra|val1|val2|val3|`.

其中每一项的意义如下表所示:

项名	意义
"	开始一个注释
ID	一个标识符, 脚本可以用它来区分不同的配置组. 当脚本启动时, 用户要告诉脚本应该处理哪个配置组
tabname	应用配置的 tab 的名字 (在脚本中称为 leaf)
text	和配置相关的文本信息
command	配置项所要执行的实际命令, 比如 <code>set textwidth=</code> , 注意, 不需要填上具体的值
extra	比如说, 如果命令是一个映射, 则变量部分就是被映射的按键, 这时候用户还需要设置映射到的命令, 命令就是填在 extra 字段中, 如果不需要它, 就留空
val1-val3	命令参数的默认值, 在配置系统中可以按 Tab 键来遍历这三个值.

219

再在来看一个例子:

```
"|SETTINGS|Layout|Set the text width|set tw=|50|70|90|
set tw=60
"|SETTINGS|Layout|Show a ruler?|let &ruler=||0|1|
let &ruler=1
```

这个例子展示了设置 Vim 的两种典型方法 — 用 set 与 let 来设置一个选项. 在变量名左边加上 & 是在告诉 Vim 等号右边的表达式要在设置之前求值, 如果求值结果的类型和选项不一致, 还会自动转换类型. 注释下面是真正的设置命令.

如果只是想在 tab 中添加一个注释, 只需要把命令替换成 %, 比如:

```
"|SETTINGS|Layout|Here you will find layout settings|%
```

用户还可以利用它来插入空行, 这些空行将作为间距来使用, 方法是用空格替换掉文本.

接下来讨论如何设置脚本.

设置脚本的地方在 setup.vim, 在脚本中搜索 HERE2, 找到该单词出现的第二个地方, 你将会看到:

```
" Import settings from these files HERE2:
let setup_files=Arr("settings.vim", "mappings.vim", "scripts.vim")
let setup_group="SETTINGS"
```

setup_files 是一个数组, 数组中的每一个元素都是一个文件名, 文件包含了设置信息. 上面的例子用到了 settings.vim, mappings.vim, scripts.vim 这三个文件.

220

第二条命令告诉脚本当前设置使用哪一个配置组。在示例中用的是 `SETTINGS`，因此如果某个设置的注释以 `|SETTINGS|` 开始，那么这个设置就会被用到。

在开始使用配置向导之前，在 `setup.vim` 中还有一项需要修改。

由于设置分散在多个文件中，我们需要让 Vim 意识到这点，因此 `vimrc` 要包含（在 Vim 中称为 `source`）这些文件，除此之外，为了让修改生效，还要让 Vim 重新加载 `vimrc`，为了处理这种情况，脚本提供了一个特殊的设置。

在 `setup.vim` 中搜索 `HERE1`，找到单词出现的第二个地方，用户将会看到：

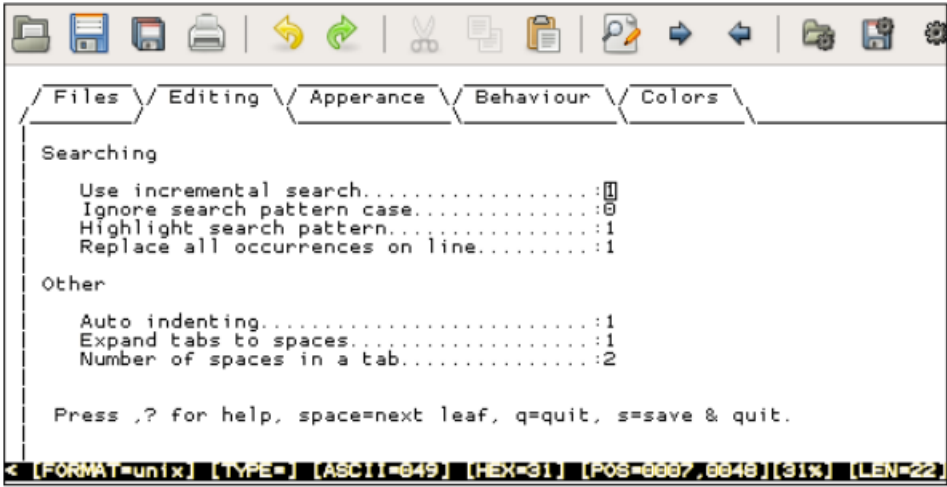
```
autocmd bufleave _setup source ~/.vimrc
```

当用户离开配置向导时就会执行 `source ~/.vimrc`。如果需要的话，用户还可以添加其他外部配置文件。

现在讨论如何启动 Vim 配置向导，并在向导中导航。为了启动配置向导，先切换到普通模式，然后按下 `s`，启动后，光标位于第一个 `tab` 的第一个设置上。在向导中有很多快捷键可以使用：

快捷键	描述
r 或 R	进入替换模式
s	保存后退出
q	不保存退出
<space>	遍历所有的 <code>tab leaf</code>
j	跳转到当前 <code>tab leaf</code> 的下一个设置
k	跳转到当前 <code>tab leaf</code> 的前一个设置
<tab>	显示被选中的设置的可选值
<cr>	接收当前行的修改
<esc>	重新绘制当前的 <code>tab leaf</code> （普通模式）
<esc>	丢弃修改（插入或替换模式）

使用配置向导的准备工作就是这些。下面的截图展示了向导的使用效果：



B.3 在线存放 vimrc

假设用户要在多种不同的计算机中使用 Vim，可能会因为每次都要重新设置 Vim，或者使用不同的设置而感到很恼火。

现在, 几乎每一台计算机都连接到了因特网上, 所以一个明显的解决办法是在线保存一份 vimrc. 之前已经说过 Vim 可以编辑远程主机上的文件, 但是, 如果要读取远程主机上的 vimrc 将会遇到一些问题. 通常情况下, 当 Vim 需要读取网络上的文件时, 它会用到插件 netrw, 然而, 当 Vim 读取 vimrc 时, 这个插件还没有加载, 也就无法读取远程主机上的 vimrc.

解决办法是使用下面的函数:

```
function! GetNetVimrc(vimrc_url)
    source $VIMRUNTIME/plugin/netrwPlugin.vim
    Nread a:vimrc_url
    let tmpfile = tempname()
    save! tmpfile
    source tmpfile
    delete(tmpfile)
    enew
endfunction
```

读取 vimrc 的所有操作都在 GetNetVimrc 中完成. 函数的输入参数是 vimrc 的 URL 地址, 函数开始时先加载 netrw 插件, 加载后 Vim 就可以使用网络读写功能, 然后, 调用 Nread 从网络上读取 vimrc 到当前缓冲区, 再把文件的内容写到一个临时文件, 加载它, 最后再把临时文件删掉, 并打开一个干净的缓冲区.

那么, 用户应该如何使用 GetNetVimrc?

首先把 vimrc 保存到网络上的某台主机, 如果用户想要在某个系统中使用自己的在线 vimrc, 就把 GetNetVimrc 添加到系统本地的 vimrc.

调用函数的方式是:

```
:call GetNetVimrc("http://www.domain.com/myvimrc")
```

还可以把上面这行代码添加到包含 GetNetVimrc 的 vimrc 文件中.

现在, 用户所有系统中的 Vim 都具有了相同的配置, 如果在线的 vimrc 有了更新, 那么等下一次用户使用系统时, 这些更新也会反映在系统中.

记住, 用户还可以用 Nread 和 Nwrite 这两个函数来直接修改在线 vimrc.

附录 C 索引

索引中的页码指的是英文原版的页码, 与本书页边标注的页码一致.

符号

:diffsplit 112
:diffthis 112
:pop 82
:ptselect 82
:tag 82
:tnext 82
:tprev 82
:tselect 82
:vert diffsplit 112
<cWROD> 61
<Leader> 178
<Plug> 178
<silent> 31
<unique> 178

A

AbbrAsk 49
amenu 33
argnum 171
Aspell 40
autocompletion
 about (关于) 84
 all-in-one completion (多合一补全) 89, 90
 dictionary completion (字典补全) 85,86
 known word completion (已知单词补全) 84,85
 omnicompletion 86-88
autoindent 130

autoprotovim 84

B

balloons 43
Berkeley Par, external formatting tools (外部格式化工具) 137
black hole register (黑洞寄存器) 102
buffers 32

C

calcValue 81
charityware license (慈善授权) 15
cindent
 about (关于) 131
 setup options (配置选项) 131
code block formatting (代码块格式化)
 commands (命令) 133
code, formatting (代码格式化)
 about (关于) 129
 autoindent 130
 cindent 131
 code-block, formatting (代码块格式化) 132-134
 indentexpr 132
 pasted code, auto formatting (自动格式化粘贴的代码) 135
 settings (设置) 130
 smartindent 130
color scheme (配色方案) 147

color scheme, Vim (Vim 的配色方案)
 changing (修改) 21
configuration files (配置文件)
 exrc 19
 gvimrc 19
 types (配置文件的类型) 18
 vimrc 18,19
context-aware navigation (基于上下文的导航)
 about (关于) 54
 code file, moving around within (在代码文件内导航)
 54, 55
 code file, moving in (在代码文件内导航) 56-58
ctags command-line program (ctags 命令行程序) 80
ctags.vim 83
cURL 213
cursorline 38

D

debugger commands (调试器命令)
 about (关于) 187
 cont 187
 finish 187
 interrupt 187
 next 187
 quit 187
 step 187
debugging (调试)
 Vim scripts (Vim 脚本) 186-188
drop registers (投递寄存器) 102

E

editor area, Vim (Vim 的编辑器区域)
 abbreviations, using (使用缩写) 46-48
 key bindings, modifying (修改按键绑定) 49, 51
 line numbers, adding (添加行号) 39
 personalizing (个性化) 37
 spell check (拼写检查) 40-42
 tooltip, adding (工具提示) 43-46

editor area, Vim visual cursor, adding (添加可视化光标) 37, 38

Elvis

 about (关于) 10
 features (特性) 10

Emacs editor (Emacs 编辑器) 13

expression register (表达式寄存器) 103

exrc 19

external formatting tools (外部格式化工具)

 about (关于) 136

 Berkeley Par 137

 Indent 136

 Tidy 138

 using (使用方法) 136

external interpreter (外部解释器)

 using in Vim scripting (在 Vim 脚本中使用) 194

F

file explorer (文件浏览器) 208

file navigation (文件导航)

 about (关于) 54

 context-aware navigation (基于上下文的导航) 54

 long line, navigating (长行导航) 59

File-Type plugins group (文件类型插件组) 148

fold (折叠)

 about (关于) 107

 diff, using to track changes (使用 diff 跟踪差异)
 114

 simple text file outlining (简单的文本文件提纲) 110,
 111

 types (类型) 107

 using (使用方法) 107-109

 vimdiff, using to track changes (使用 vimdiff 跟踪差异) 111

foldclose() 45

fonts, Vim (Vim 字体)

 changing (修改) 20

for loop (for 循环) 164, 165

formatexpr 123

formatting, Vim (Vim 格式化)
 code, formatting (代码格式化) 129
 external formatting tools, using (使用外部格式化工具) 136
 text, formatting (文本格式化) 121
functions (函数)
 creating (创建) 168-170
 variable argument list (可变参数列表) 170-172

G

Game of Life (生命游戏) 202
get 163
Global plugins group (全局插件组) 148
guitablabel 属性 36
gvimrc 19

H

hacker (黑客) 15
hacking 15
hasmapto() 178
helpgrep 67
hidden markers (隐藏标记)
 about (关于) 71
 marks, using (使用标记) 71

I

IDE (集成开发环境) 208
indentexpr 132
Indent, external formatting tools (外部格式化工具) 136
integrated compiler (集成的编译器) 207
integrated debugger (集成的调试器) 207
interpreter (解释器) 15 Ispell 40

J

join 163

K

key bindings, editor area (按键绑定)
 modifying (修改) 49, 51
keys() 165
keyvar 165

L

line numbers, editor area (行号)
 adding (添加) 39
loaded_myscript 176
LoadTemplate 77
lookupfile.vim 83
loops (循环)
 about (关于) 164
 for loop (for 循环) 164
 types (类型) 164
 while loop (while 循环) 164

M

macro recording (宏录制)
 about (关于) 90
 using (使用方法) 90-92
map 30, 163
marks (书签) 71
matching features, Vim (Vim 的匹配特性) 22
menu 30
MicroEmacs code (MicroEmacs 的代码) 13
Mines (扫雷) 204
MS Visual Studio® 207
Mutt 210
mydict dictionary (字典 mydict) 165
MyIndenter() 132

N

named registers (命名寄存器) 101
 navigation, in multiple buffers (在多个缓冲区中导航)
 61
 navigation, in Vim help (在 Vim 帮助系统中导航) 60
 Nibbles (贪吃蛇) 202
 numbered registers (编号的寄存器) 100
 nvi
 about (关于) 10
 features (特性) 11

O

omnicompletion 86-88

P

Perl 195
 personal highlighting, Vim (Vim 的修改化高亮)
 about (关于) 22, 23
 color character, marking (彩色字符标记) 24
 errors, preventing (抑制错误) 26
 tabs not used for indentation, marking (标记不是用作
 缩进的制表符) 25
 PrintSum 169
 project browser (项目浏览器) 208
 Python 196

R

range() 164
 read-only registers (只读寄存器) 101
 registers (寄存器)
 about (关于) 98
 drop registers (投递寄存器) 102
 expression registers (表达式寄存器) 103
 named registers (命名寄存器) 101
 numbered registers (带编号的寄存器) 100
 read-only registers (只读寄存器) 101
 search pattern register (搜索模式寄存器) 102

selection register (选择寄存器) 102
 small delete register (小删除寄存器) 100
 unnamed register (匿名寄存器) 100
 using (使用方法) 99

remote files (远程文件)
 editing (编辑) 117, 118
 working in (处理远程文件) 115-117
 Rubik's cube (魔方) 203
 Ruby 198

S

script (脚本) 15
 script development (脚本开发)
 about (关于) 150, 151
 script writing basic (脚本开发基础) 151
 scripting tips, Vim (Vim 脚本开发技巧)
 Gvim, using (使用 Gvim) 182
 longer lines, printing (打印长行) 185
 multiple operating system, working with (处理多个操
 作系统) 183
 versions, of Vim (Vim 的版本) 183
 script writing basics (脚本开发基础)
 about (关于) 151
 conditions (条件) 157, 158
 dictionaries, working with (处理字典) 159-163
 functions, creating (创建函数) 168
 lists, working with (处理线性表) 159-163
 loops (循环) 164
 types (类型) 152
 variables (变量) 153-157
 scrollbind 113
 search pattern register (搜索模式寄存器) 102
 search, Vim (Vim 搜索)
 examples (例子) 64, 65
 help system, searching (在帮助系统中搜索) 67
 searching, in current file (在当前文件中搜索) 64
 searching, in multiple files (在多个文件中搜索) 65,
 66
 selection registers (选择寄存器) 102

sessionoptions

- blank 96
- buffers 96
- curdir 96
- folds 96
- globals 96
- help 96
- localoptions 96
- options 96
- resize 96
- sesdir 96
- slash 96
- tabpages 96
- unix 96
- winpos 96
- winsize 96

sessions (会话)

- sessionoptions 96
- simple session usage (会话的简单用法) 93-95
- using (使用方法) 93
- using, as project manager (项目管理程序) 97, 98

setup options, cindent (cindent 的设置选项)

- cinkeys 131
- cinoptions 131
- cinwords 131

ShortTabLine() 35

sign 68

Single Unix Specification (SUS, 单一 Unix 规范) 9

small delete register (小删除寄存器) 100

smartindent 130

snipMate plugin (snipMate 插件) 79

snipMate system (snipMate 系统) 79

Sokoban (推箱子) 205

sort() 165

spellllang 40

spellsuggest() 45

split 163

status line, Vim (Vim 的状态行) 26, 28

STEVIE 9

suffixadd 62

sum 170

syntax coloring (语法高亮) 142, 143, 147

syntax-color scheme (语法高亮主题) 141

syntax regions (语法区) 143-146

T

tabs, Vim (Vim 的标签页)

- modifying (修改) 33-37

tag browser (Tag 浏览器) 208

tag list generators (Tag list 生成程序)

- about (关于) 80

- Ctags 80

- Hdrtags 80

- Jtags 80

- Ptags 80

- Vtags 80

tag lists

- about (关于) 80

- taglist navigation (taglist 导航) 83

- uses 83

- using (使用方法) 80-82

taglist.vim 83

templates (模版)

- abbreviations, using (使用缩写) 76, 77

- about (关于) 74

- snippets, with snipMate script (通过 snipMate 使用代码片断) 78, 79

- template files, using (使用模版文件) 74, 75

Tetris (俄罗斯方块) 206

text, formatting (文本格式化)

- about (关于) 121

- headlines, marking (标题行标记) 125, 126

- lists, creating (创建线性表) 127-129

- text, aligning (文本对齐) 124

- text, putting into paragraph (文本分段) 122, 123

The Mail Suite (TMS, 邮件套装) 210

Tic-Tac-Toe (井字棋) 204

Tidy, external formatting tools (外部格式化工具 Tidy)

TwitVim 212

U

undo branching (撤消分支)

about (关于) 98

using (使用方法) 103-106

unnamed register (匿名寄存器) 100

V

variables (变量)

about (关于) 153

dictionary (字典) 153

funcref 153

list (线性表) 153

number (数值) 153

string (字符串) 153

`v:folddashes` 109

`v:foldend` 109

`v:foldstart` 109

`vi` 9

vi compatibility (vi 兼容性) 14, 15

Vile

about (关于) 13

features (特性) 13

Vim

about (关于) 7, 11

advanced formatting (高级格式化) 121

autocompletion (自动补全) 84

charityware license (慈善授权) 15

color scheme, changing (修改配色方案) 21

command line buffer (命令行寄存器) 26

configuration files (配置文件) 18

download link (下载链接) 8

editor area, personalizing (个性化的编辑区) 37

extensibility (可扩展性) 141

features (特性) 12

fonts, changing (修改字体) 20

hidden markers (隐藏标记) 71

mail program (邮件程序) 210

marks, adding (添加标记) 68

matching (匹配) 22

menu, adding (添加菜单) 29-32

menu, toggling (切换菜单) 28, 29

personal highlighting (个性化高亮) 22, 23

personalizing (个性化) 17

scripting tips (脚本开发技巧) 182

script structure (脚本结构) 175

search (搜索) 63

status line (状态行) 26

syntax-color schemes (语法配色方案) 141

tabs, modifying (修改标签页) 33-37

toolbar icons, adding (添加工具栏图标) 32, 33

toolbar, toggling (切换工具栏) 28, 29

using, as Twitter client (Twitter 客户端) 212

visible markers (可视化的标记) 68-70

Vimballs

creating (创建) 190

vimdiff

about (关于) 112

navigation (导航) 113

using, to track changes (跟踪差异) 111

vimdiff session (vimdiff 会话) 112

Vim documentation (Vim 文档) 191, 193

Vim games (Vim 游戏)

about (关于) 201

Game of Life (生命游戏) 202

Mines (扫雷) 204

Nibbles (贪吃蛇) 202

Rubik's cube (魔方) 203

Sokoban (推箱子) 205

Tetris (俄罗斯方块) 206

Tic-Tac-Toe (井字棋) 204

VimIRC 211, 212

vimrc

about (关于) 18, 19

cleaning, tips (清理技巧) 215

online storing (在线存储) 221

vimrc file, cleaning tips (vimrc 清理技巧)

- comments, using (使用注释) 216
- data, grouping (数据分组) 216
- multiple files, using (使用多个文件) 216
- Vim, using in noncompatible mode (在非兼容模式中使用 Vim) 216
- vimrc setup system (vimrc 设置系统)
- Vim script (Vim 脚本)
 - debugging (调试) 186-188
 - distributing (发布) 189
 - external interpreters, using (外部解释器) 194
 - installing (安装) 148, 149
 - scripting tips (脚本开发技巧) 182
 - structure (结构) 175
 - types (类型) 148
 - uninstalling (卸载) 150
 - using (使用方法) 147
- Vim scripting, in Perl (使用 Perl 开发 Vim 脚本) 195, 196
- Vim scripting, in Python (使用 Python 开发 Vim 脚本) 196, 197
- Vim scripting, in Ruby (使用 Ruby 开发脚本) 198, 199

- Vim script structure (Vim 脚本结构)
 - about (关于) 175
 - functions (函数) 178, 179
 - key mappings (按键映射) 178, 179
 - script configuration (脚本配置) 177
 - script header (脚本头部信息) 176
 - script-loaded check (脚本加载检查) 176
- visible markers (可视化标记)
 - about (关于) 68-70
 - sign, using (使用 sign) 68-70
- visual cursor, editor area (可视化光标)
 - adding (添加) 37

W

- while loop (while 循环) 166, 167

X

- xvile 13