

目录

前言:	1
第一部分: vim 基本操作.....	2
第二部分: 配置 vim.....	5
1、安装 Vim 和 Vim 基本插件.....	6
2、Vim 配置文件.....	7
3、ctags 安装与配置及 vimgrep.....	8
4、使用 taglist 插件.....	10
5、vim 智能补全.....	12
6、文件浏览和缓冲区浏览.....	17
7、使用 lookupfile 插件.....	21
8、集成编译 GCC 和调试 GDB.....	26
总结.....	35

作者: **kelly_Lok**

联系方式(QQ/微信):

746768750

前言：

所谓的高手，就是在某个地方比你多看多三两眼，确实如此。我们不是要成为 vim 的顶尖高手，我们只是想利用 vim 提高编程效率而已，因此，不需要记住那么多快捷键，我们只需要记住常用的快捷键就行。感谢杨源所著的《vim 高级使用》，让我学到很多 vim 额外的知识，感谢开发 vim 插件的大神们的辛勤劳动，希望本文可以起到抛砖引玉的作用，给对 vim 感兴趣的读者一个参考。

第一部分：vim 基本操作

在 vim 中执行“:help usr_02.txt”可以查看使用手册，本章参考 vim 的使用手册，对应具体的使用，有兴趣的读者可以参考以上命令查看具体的使用手册：

初步知识

[|usr_01.txt|](#) 关于本手册

[|usr_02.txt|](#) Vim 初步

[|usr_03.txt|](#) 移动

[|usr_04.txt|](#) 做小改动

[|usr_05.txt|](#) 选项设置

[|usr_06.txt|](#) 使用语法高亮

[|usr_07.txt|](#) 编辑多个文件

[|usr_08.txt|](#) 分割窗口

[|usr_09.txt|](#) 使用 GUI 版本

[|usr_10.txt|](#) 做大修改

[|usr_11.txt|](#) 从崩溃中恢复

[|usr_12.txt|](#) 小窍门

高效的编辑

[|usr_20.txt|](#) 快速键入命令行命令

[|usr_21.txt|](#) 离开和回来

[|usr_22.txt|](#) 寻找要编辑的文件

[|usr_23.txt|](#) 编辑特殊文件

[|usr_24.txt|](#) 快速插入

|usr_25.txt| 编辑已经编排过的文本

|usr_26.txt| 重复

|usr_27.txt| 查找命令及模式

|usr_28.txt| 折叠

|usr_29.txt| 在代码间移动

|usr_30.txt| 编辑程序

|usr_31.txt| 利用 GUI

|usr_32.txt| 撤销树

调节 Vim

|usr_40.txt| 创建新的命令

|usr_41.txt| 编写 Vim 脚本

|usr_42.txt| 添加新的菜单

|usr_43.txt| 使用文件类型

|usr_44.txt| 自定义语法高亮

|usr_45.txt| 选择你的语言

让 Vim 工作

|usr_90.txt| 安装 Vim

Vim 常用的几种模式：

n	Normal mode
v	Visual mode
i	Insert mode
c	Command-line mode

Esc 键可以在任意模式切换到命令模式

以‘:’、’/’、’?’操作的，我把它称为底部模式

使用插入命令，可进入插入模式，也称编辑模式

1.1 移动命令

- 1、按字符：H,J,K,L——左，下，上，右
- 2、按单词：w(向前)，b(往后)
- 3、翻页：CTRL-F(上一页)，CTRL-B(下一页)

4、行首： `0`(数字 0，不是字母 o)，或者 `CTRL-A`

5、行尾： `$`，或者 `CTRL-E`

前面三种都可以配合 `n` 使用，`n` 为数字，代表重复执行 `n` 次，如“`5w`”表示向前移动 5 个单词。

1.2 插入命令

光标前： `i`

光标后： `a`

光标上一行： `O`(大写字母)

光标下一行： `o`(小写字母)

光标所在行首： `I`

光标所在行尾： `A`

1.3 复制命令

整行复制： `yy`

从光标所在行开始复制 `n` 行： `nyy`

粘贴： `p`

1.4 撤销命令

撤销： `u`

反撤销： `CTRL-R`

1.5 保存退出

退出： `":q"`

保存： `":w"`

强制退出： `":q!"`

保存所有： `":wa"`

保存退出： `":wq"`或者 `ZZ`

1.6 查找命令

`/` : 往下查找

`?` : 往上查找

`N` : 执行查找命令后，查找上一个

n : 执行查找命令后, 查找下一个

1.7 文档对齐

对齐: **G=gg**

1.8 屏幕分割

1、在 shell 执行: **vim -On filename**

-O 垂直分屏

-o 水平分屏

n 数字, 切割 **n** 个屏

2、屏的切换: **CTRL-W-W** 或者配合 **H,J,K,L** 使用(**CTRL-W-H**)

1.9 标签页

1、新建标签页: **":tabnew"**

2、标签页的切换: **gt**

1.10 tag 跳转

1、**":tag {ident}"**

2、跳转下一标签: **CTRL-]**

3、跳转上一标签: **CTRL-T**

1.11 系统调用:

跳转到系统函数的说明: **K**

光标停在调用的系统函数上, 按"**K**"键, 就会在当前页面打开该函数的详细信息, 跟在 shell 用 **man** 命令查看手册是一样的。"**q**"键退出并返回编辑页面。

第二部分: 配置 vim

问题 1: 为什么直接复制 **/etc/vim/vimrc** 的文件到 **/home/user/** 目录下, 切换到该用户的根目录就能使用 **.vimrc** 配置文件?

答: 因为存在环境变量 **%HOME**, 记录着每个用户的根目录, 当切换到普通用户, 使用 **vim** 的时候, **vim** 会优先在 **HOME** 路径寻找 **vimrc**, 因此, 我们只需要修改当前用户的 **vimrc** 配置文件即可。

问题 2: 普通用户, 在 **root** 用户的文件目录使用 **vim** 创建新文件, 换言之,

在隶属于 root 用户的目录下，使用了 vim 命令创建新文件，忘了加 sudo。则在保存文件的时候，提示出错。强制退出的话，原来的文本就丢失，怎么解决？

答：1、使用命令重定向：`:w !sudo tee % > path/filename` (注意重定向的路径必须是当前用户有权限访问的)

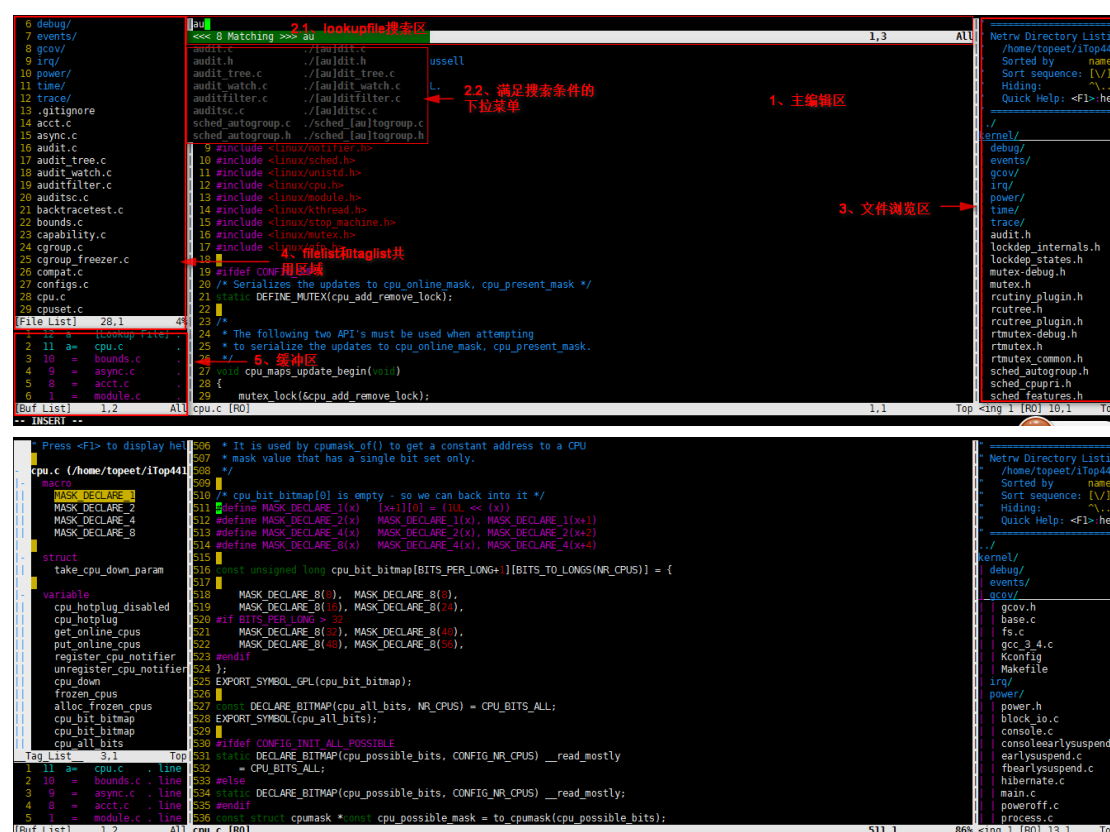
2、执行命令：`q!` 强制退出

问题 3：用 vim 打开含有中文的文件，出现乱码，怎么办？

答：打开当前用户的 `vimrc` 配置文件，添加：

`set fencs=utf-8,GB18030,ucs-bom,default,latin1`

vim 最终效果图：启动了 winManager (TagList, FileExplorer, BufExplorer)、lookupfile 和 vim 自带的 netrw



1、安装 Vim 和 Vim 基本插件

首先安装好 Vim 和 Vim 的基本插件。这些使用 apt-get 安装即可：

```
topeet@ubuntu:~$ sudo apt-get install vim vim-scripts vim-doc
```

其中 vim-scripts 是 vim 的一些基本插件，包括语法高亮的支持、缩进等等。

vim 中文帮助文档 tar 包下载地址：

<http://sourceforge.net/projects/vimcdoc/files/vimcdoc/>

若不能弹出下载页面，可以在左上角搜索：vimcdoc，本文所涉及的 vim 官网提供的插件链接，若打开链接没有下载页面，可以在页面的左上角的搜索栏所搜插件名称进入下载页面。

下载：vimcdoc-1.9.0.tar.gz

将其解压缩：

```
topeet@ubuntu:~$ tar -zxvf vimcdoc-1.9.0.tar.gz
```

然后进入目录：

```
topeet@ubuntu:~$ cd vimcdoc-1.9.0/
```

切换到 root 用户：

```
topeet@ubuntu:~/vimcdoc-1.9.0$ su root
```

执行：

```
root@ubuntu:/home/topeet/vimcdoc-1.9.0# ./vimcdoc.sh -i
```

建议以 root 身份安装，当以 root 身份安装时，文件会被拷贝至 /usr/share/vim/vimfiles/doc 下。因此所有系统的用户都可以使用中文文档。如果你的 vim 是安装在 /usr/local 下的话，你需要这样设定 vim 的 runtimepath 选项：

```
:set rtp+=/usr/share/vim/vimfiles
```

2、Vim 配置文件

Vim 的系统配置文件夹是在 /usr/share/vim/ 和 /etc/vim/ 这两个文件夹下。一般情况下，我们不会去改变这两个文件夹下的配置文件，而是在用户根目录建立自己的配置文件。我的用户名是 topeet，对应用户根目录为：/home/topeet/，为了方便使用，我们将示例的 vimrc 拷贝到用户根目录下。示例的 vimrc (名为 vimrc_example.vim) 通常位于 /usr/share/vim/vimXXX/ 目录下，其中 vimXXX 与你所使用的 vim 版本有关。

执行以下 shell 命令：

```
topeet@ubuntu:~$ cp /usr/share/vim/vim73/vimrc_example.vim ~/.vimrc
```

或者在 vim 中执行：

```
:!cp $vimRUNTIME/vimrc_example.vim ~/.vimrc
```

以上两条命令完成的功能是一样的,可以根据个人喜好选择。然后退出 vim,再次进入 vim,就可以看到不一样的效果了。

3、ctags 安装与配置及 vimgrep

Tag 文件(标签文件)无疑是开发人员的利器之一,有了 tag 文件的协助,你可以在 vim 查看函数调用关系,类、结构、宏等的定义,可以在任意标签中跳转、返回.....相信使用过 Source Insight 的人对这些功能并不陌生,而在 vim 中,此功能的实现依赖于 tag 文件。

通常我们使用名为 ctags 的程序来生成这样的 tag 文件,vim 能直接使用 ctags 程序所生成的 tag 文件。在 UNIX 系统下的 ctags 功能比较少,所以一般我们使用 Exuberant Ctags,在 ubuntu 下 ctags 的下载安装和配置过程:

下载并安装 ctags,终端输入命令:

```
topeet@ubuntu:~/vimcdoc-1.9.0$ sudo apt-get install ctags
```

或者到官网下载源文件编译: <http://ctags.sourceforge.net/>

源文件编译的步骤:

- 1、解压: `$ tar -zxvf ctags-5.8.tar.gz`
- 2、切换目录: `$ cd ctags-5.8`
- 3、配置: `$. /configure --prefix=~/.vim/ctags` (可以指定安装目录,也可以不指定,默认安装在/usr/local)
- 4、`$ make`
- 5、`# make install` // 需要 root 权限
- 6、若要在任意的目录使用,需要创建一个软链接到/usr/bin/,或者将安装路径加入到系统环境变量中。

```
topeet@ubuntu:~/ctags-5.8$ sudo ln -s ~/.vim/ctags/bin/ctags /usr/bin/ctags
```

虽然 ctags 有为数众多的选项,但通常我们所使用的非常简单。以 ctags 的源代码为例:

- 1、切换到 ctags 源代码目录:


```
topeet@ubuntu:~/vim/ctags/bin$ cd ~/ctags-5.8/
```

2、执行：

```
topeet@ubuntu:~/ctags-5.8$ ctags -R *
```

其中，-R 参数代表递归子目录，*代表所有文件，输入命令 ctags -R *，可以发现多了一个 tags 文件，这个就是索引文件。

```
topeet@ubuntu:~/ctags-5.8$ ls
ant.c      config.h.in  e_msoft.h   html.c      MAINTAINERS  options.c    readtags.h  sql.c
ant.o      config.log   entry.c     html.o      make.c       options.h    readtags.o  sql.o
argproc.c  config.status entry.h     INSTALL     Makefile     options.o    rexx.c      strlist.c
args.c     configure    entry.o     INSTALL.oth Makefile.in  parse.c     rexx.o      strlist.h
args.h     configure.ac e_os2.h     jscript.c   make.o       parse.h     routines.c  strlist.o
args.o     COPYING     e_qdos.h    jscript.o   matlab.c     parse.o     routines.h  tags
asm.c      ctags       e_riscos.h  keyword.c   matlab.o     parsers.h   routines.o  tcl.c
asm.o      ctags.1     erlang.c    keyword.h   mk_bc3.mak   pascal.c   ruby.c      tcl.o
asp.c      ctags.h     erlang.o    keyword.o   mk_bc5.mak   pascal.o    ruby.o      tex.c
asp.o      ctags.html  e_vms.h     lisp.c      mk_djg.mak   perl.c     scheme.c    tex.o
awk.c      ctags.spec  EXTENDING.html lisp.o     mk_installdirs perl.o      scheme.o    verilog.c
awk.o      debug.c     FAQ         lregex.c    mk_manx.mak  php.c      sh.c        verilog.o
basic.c    debug.h     flex.c      lregex.o    mk_mingw.mak php.o       sh.o        vhd1.c
basic.o    descrip.mms flex.o      lua.c       mk_mpw.mak   python.c   slang.c     vhd1.o
beta.c     dosbatch.c  fortran.c   lua.o       mk_mvc.mak   python.o    slang.o     vim.c
beta.o     dosbatch.o  fortran.o   mac.c       mk_os2.mak   qdos.c     sml.c      vim.o
c.c        e_amiga.h   general.h   magic.diff  mk_qdos.mak  read.c     sml.o      vstring.c
c.o        e_djgpp.h   get.c       main.c      mk_sas.mak   read.h     sort.c      vstring.h
cobol.c    eiffel.c    get.h       main.h      NEWS         README     sort.h      vstring.o
cobol.o    eiffel.o    get.o       main.o      ocaml.c      read.o     sort.o      yacc.c
config.h   e_mac.h     gnu_regex   maintainer.mak ocaml.o      readtags.c source.mak  yacc.o
```

现在我们进入 vim(在终端输入 vim 回车)，执行下面的命令：

```
:cd ~/ctags-5.8 "切换到 tags 所在目录
```

```
:set tags=tags "设置 tags 选项为当前目录下的 tags 文件
```

这样，我们设置好了 tags 选项，接下来我们使用它：

```
:tag main
```

可以看到 vim 打开了 eiffel.c 文件，并把光标定位到第 1292 行 main 上。但是，这并不是我们所期待的，一般主程序的函数名为 main，并且在 main.c 这个文件中。这是因为 ctags 并不是编译器，它在处理编译预处理指令受到局限，因此并没有生成 main.c 中 main() 函数的标签。

我们可以用“:vimgrep”来查找 main 函数：

```
:vimgrep /\<main\>/ ./*.c
```

```
:cw
```

执行 :cw 打开 quickfix 窗口，这时下面的 quickfix 窗口将显示出来，在 quickfix 窗口中找到我们想跳转的位置(本例中是第 8 行的 ./main.c)，按回车，就可以跳到对应的位置了。

```
525 extern int main (int __unused __argc, char **argv)
526 {
527     cookedArgs *args;
528 #ifdef VMS
529     extern int getredirection (int *ac, char ***av);
530
531     /* do wildcard expansion and I/O redirection */
./main.c 525,12 91%
1 ./argproc.c|77 col 5| *      main (argc, argv)
2 eiffel.c|1292 col 12| extern int main (int argc, char** argv)
3 ./entry.c|44 col 11| #include "main.h"
4 ./fortran.c|2092 col 5| /*  main-program is
5 ./fortran.c|2105 col 12| *      is main-program
6 ./main.c|2 col 10| *      $Id: main.c 536 2007-06-02 06:09:00Z elliotth $
7 ./main.c|81 col 11| #include "main.h"
8 ./main.c|525 col 12| extern int main (int __unused __argc, char **argv)
9 ./options.c|24 col 11| #include "main.h"
10 ./parse.c|22 col 11| #include "main.h"
[Quickfix List] :vimgrep /\<main\>
```

在我的 vimrc 中，定义下面的键映射，利用它可以在当前文件中快速查找光标下的单词：

```
nmap <leader>lv :lv /<c-r>=expand("<cword>")<cr>/ %<cr>:lw<cr>
```

vim 会保存一个跳转的标签栈，以允许你在跳转到一个标签后，再跳回来，可以使用“:tags”命令查找你处于标签栈的哪个位置。

我们经常用到的 tag 跳转命令在第一部分已经介绍了。

由于我们配置的 vimrc 是独立于系统配置，因此需要建立自己的 vim 配置文件。在 ~/.vim 目录下，新建两个子目录：

```
topeet@ubuntu:~$ cd ~/.vim
topeet@ubuntu:~/.vim$ mkdir plugin doc
```

其中，plugin 目录存放插件，doc 目录存放插件的帮助文档。

4、使用 taglist 插件

现在我们到 <https://sourceforge.net/projects/vim-taglist/files/> 下载最新版本的 taglist plugin，目前版本是 4.6。

下载后，把该文件在 ~/.vim/ 目录中解压缩：

```
topeet@ubuntu:~/.vim$ unzip taglist_46.zip
```

解压后会自动在你的 ~/.vim/plugin 和 ~/.vim/doc 目录中各放入一个文件：
plugin/taglist.vim -taglist 插件

doc/taglist.txt - taglist 帮助文件

要使用 taglist plugin，必须满足：

- 打开 vim 的文件类型自动检测功能: filetype on
- 系统中装了 Exuberant ctags 工具, 并且 taglist plugin 能够找到此工具 (因为 taglist 需要调用它来生成 tag 文件)
- 你的 vim 支持 system() 调用

前面的操作, 我们使用了 vim 自带的示例 vimrc, 这个 vimrc 中已经打开了文件类型检测功能; 在上文中, 我们也已用到了 Exuberant ctags; system() 调用在一般的 vim 版本都会支持, 所以我们已经满足了这三个条件。

使用下面的命令生成帮助标签 (下面的操作在 vim 中进行):

```
:helptags ~/.vim/doc
```

生成帮助标签后, 你就可以用下面的命令查看 taglist 的帮助了:

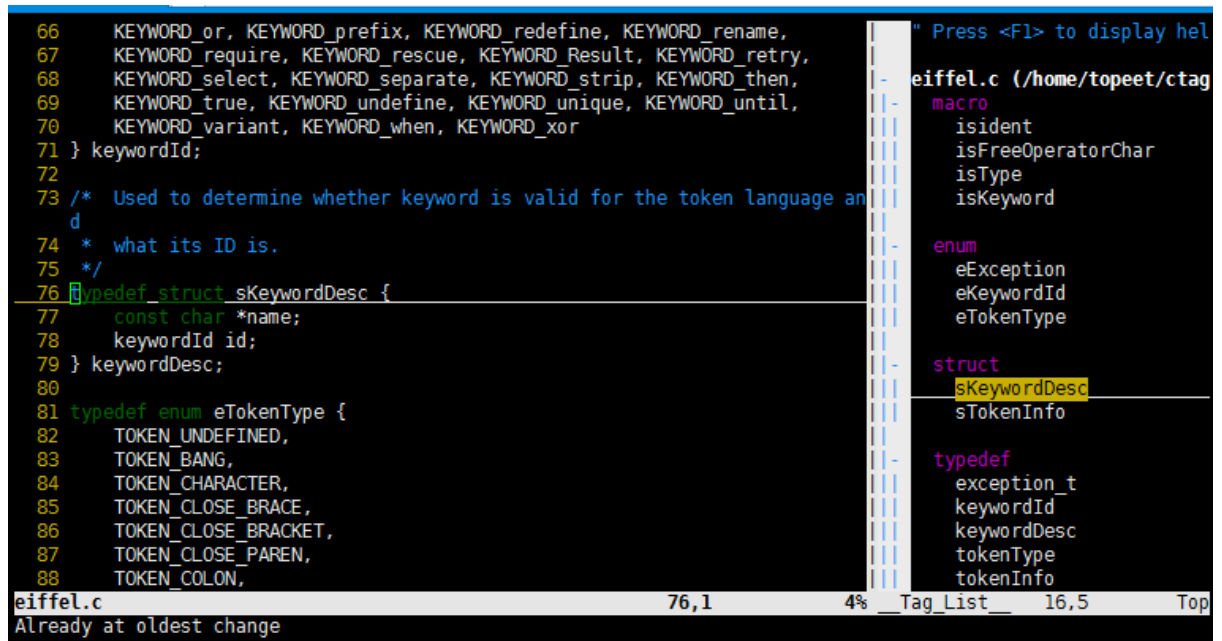
```
:help taglist.txt
```

Taglist 提供了相当多的功能, 我的 vimrc 中这样配置:

```
let Tlist_Ctags_Cmd = '/usr/bin/ctags' "设定 linux 系统中 ctags 程序的位置
let Tlist_Show_One_File = 1 "不同时显示多个文件的 tag, 只显示当前文件的
let Tlist_Exit_OnlyWindow = 1 "如果 taglist 窗口是最后一个窗口, 则退出 vim
let Tlist_Use_Right_Window = 1 "在右侧窗口中显示 taglist 窗口
"使用<F8>键打开/关闭 taglist 窗口
nnoremap <silent> <F8> :TlistToggle<CR>
```

更多的配置选项, 可以查看 taglist 的帮助文档:help taglist.txt, 可以根据自己的需求配置。

这样配置后, 当你按 F8 键, 显示如下窗口:

The screenshot shows the Vim editor interface. The main window displays a C source file named 'eiffel.c' with various keyword definitions and a struct definition for 'sKeywordDesc'. The taglist window on the right shows the results of the 'tag' command, listing symbols like 'isident', 'isFreeOperatorChar', 'isType', 'isKeyword', 'eException', 'eKeywordId', 'eTokenType', 'sKeywordDesc', and 'sTokenInfo'. The status bar at the bottom indicates the current file is 'eiffel.c', line 76, column 1, and the taglist window is showing 4% of the tags, with 16 tags listed and the 'Top' of the list is visible.

通过 CTRL-w-w 切换到 taglist 窗口，然后按 F1 键可以查看操作 taglist 窗口的命令。

5、vim 智能补全

vim 的 OMNI 补全(以下称“全能补全”)可以支持多种程序语言，包括 C、C++、XML、HTML、CSS、JAVASCRIPT、PHP、RUBY 等，详细列表请参阅“:help compl-omni-filetypes”。在本文中，主要介绍 C 及 C++的全能补全。C 代码补全需要标签文件。你应该使用 Exuberant ctags 软件，因为它会加入补全所需要的额外信息。你可以在这里找到它：<http://ctags.sourceforge.net/> 推荐使用 5.6 或以后版本。

在终端输入命令：ctags -version 可查看 ctags 版本。

对于 5.5.4 版本，你应该打上增加“typename:”字段的补丁：

<ftp://ftp.vim.org/pub/vim/unstable/patches/ctags-5.5.4.patch>

如果你想补全系统函数，可以用 ctags 生成包含所有系统头文件的标签文件：

```
topeet@ubuntu:~/$  
ctags -R -f ~/.vim/systags /usr/include /usr/local/include
```

在 vimrc 文件中，把这个标签文件增加到 'tags' 选项中：

```
set tags+=~/.vim/systags
```

```
topeet@ubuntu:~/vim$ vi test.c
```

```

1 class      gnu_pbds::detail::PB_DS_CLASS_C_DEC
2 cmd        print_node_pointer(Const_Node_Iterator nd_it, integral_constant<int,false>)
3 filename    /usr/include/c++/4.4.7/ext/pb_ds/detail/tree_trace_base.hpp
4 kind        f
5 name        print_node_pointer
6 static      0
[Scratch] [Preview] 1,1 Top
2 int main()
3 {
4     pri
5 } print_node_pointer(f @@(Const_Node_Iterator nd_it, integral_constant<int,false>) - /usr/i
~ print_rate d /usr/include/term.h - 221
~ print_screen d /usr/include/term.h - 344
~ print_to_string(f throw_allocator_base::@@(std::string& s) - /usr/include/c++/4.4.7/ex
~ printf(f @@( __const char * __restrict __fmt, ...) - /usr/include/bits/stdio2.
~ printf d /usr/include/bits/stdio2.h - 108
~ printf_arginfo_function t typedef int @@ ( __const struct printf_info * __info, - /usr/include/p
~ printf_arginfo_size_function t typedef int @@ ( __const struct printf_info * __info, - /usr/include/p
~ printf_function t typedef int @@ (FILE * __stream, - /usr/include/printf.h
~ printf_info s struct @@ - /usr/include/printf.h
~ printf_va_arg_function t typedef void @@ (void * __mem, va_list * __ap); - /usr/include/printf.
~ priority_queue f @@( _InputIterator __first, _InputIterator __last, - /usr/include/c++
~ priority_queue c class @@ - /usr/include/c++/4.4.7/bits/stl_queue.h
~ priority_queue_base_dispatch s struct @@<Value_Type, Cmp_Fn, binary_heap_tag, Allocator> - /usr/inc
test.c s struct @@ : public container_tag { }; - /usr/include/c++/4.4.7/ext/p
-- Omni completion (~^N^P) Back at original

```

缺省的情况下，打开全能补全模式，会弹出下拉菜单和一个 preview 窗口(预览窗口)来显示匹配项目，下拉菜单列出所有匹配的项目，预览窗口则显示选中项目的详细信息。打开预览窗口影响我的主编辑窗口，因此我去掉了预览窗口的显示，在 `~/.vimrc` 添加：

```
set completeopt=longest,menu
```


通过查看帮助文档，可知当自动补全下拉窗口弹出后，一些可用的快捷键：

Ctrl+P 向前切换成员

Ctrl+N 向后切换成员

Ctrl+E 表示退出下拉窗口，并退回到原来录入的文字

Ctrl+Y 表示退出下拉窗口，并接受当前选项

其他补全方式：

Ctrl+X Ctrl+L 整行补全

Ctrl+X Ctrl+N 根据当前文件里关键字补全

Ctrl+X Ctrl+K 根据字典补全

Ctrl+X Ctrl+T 根据同义词字典补全

Ctrl+X Ctrl+I 根据头文件内关键字补全

Ctrl+X Ctrl+] 根据标签补全

Ctrl+X Ctrl+F 补全文件名

Ctrl+X Ctrl+D 补全宏定义

Ctrl+X Ctrl+V 补全 vim 命令

Ctrl+X Ctrl+U 用户自定义补全方式

Ctrl+X Ctrl+S 拼写建议

要支持 C++ 的全能补全，需要到 vim 主页下载 OmniCppComplete 插件，链接如下：

http://www.vim.org/scripts/script.php?script_id=1520

搜索：OmniCppComplete，就会弹出下载页面。

下载后，把它解压到你的 ~/.vim 目录：

```
topeet@ubuntu:~/.vim$ unzip omnicppcomplete-0.41.zip
```

在 vim 执行一下命令更新帮助文档：

```
:helptags ~/.vim/doc/
```

查看 omnicppcomplete 文档：

```
:help omnicppcomplete.txt
```

确保你已关闭了 vi 兼容模式，并允许进行文件类型检测：

```
set nocp
```

```
filetype plugin on
```

接下来，使用以下命令，为当前目录下的 C++ 文件生成标签文件：

```
ctags -R --c++-kinds=+p --fields=+iaS --extra=+q .
```

或者在 ~/.vimrc 配置文件添加：

” 设置 <F3> 为生成 c++tags 的快捷键

```
map <F3> :!ctags -R --c++-kinds=+p --fields=+iaS --extra=+q .<CR>
```

在对 C++ 文件进行补全时，OmniCppComplete 插件需要 tag 文件中包含 C++ 的额外信息，因此上面的 ctags 命令不同于以前我们所使用的，它专门为 C++ 语言生成一些额外的信息，上述选项的含义如下：

--c++-kinds=+p : 为 C++ 文件增加函数原型的标签

--fields=+iaS : 在标签文件中加入继承信息(i)、类成员的访问控制信息(a)、以及函数的指纹(S)

--extra=+q : 为标签增加类修饰符。注意，如果没有此选项，将不能对类成员补全

我们可以自定义自己的补全方式，使工作更加高效。可以在 vimrc 中定义下面的键绑定，以减少按键次数：

```
inoremap <C-O> <C-X><C-O>
```

```
inoremap <C-]> <C-X><C-]>
```

```
inoremap <C-F> <C-X><C-F>
```

```
inoremap <C-D> <C-X><C-D>
```

```
inoremap <C-L> <C-X><C-L>
```

SuperTab 插件会记住你上次所使用的补全方式，下次再补全时，直接使用 TAB，就可以重复这种类型的补全。比如，上次你使用 CTRL-X CTRL-F 进行了文件名补全，接下来，你就可以使用 TAB 来继续进行文件名补全，直到你再使用上面列出的补全命令进行了其它形式的补全。这个插件在下面的链接下载：

http://www.vim.org/scripts/script.php?script_id=1643

若不能弹出下载页面，可以在左上角搜索：SuperTab

下载最新版本，然后用 vim 编辑器打开：

```
topeet@ubuntu:~/.vim$ vi supertab.vmb
```


执行命令：

```
:so % 或者 :so supertab.vmb
```

其中，`:so file` 表示从 `{file}` 里读取 Ex 命令，即 `:"` 开头的命令，并执行。`%` 为通配符。

调整 SuperTab 的缺省行为：

- `g:SuperTabRetainCompletionType` 的值缺省为 1，表示记住你上次的补全方式，直到使用其它的补全命令改变它；如果把它设成 2，表示记住上次的补全方式，直到按 ESC 退出插入模式为止；如果设为 0，表示不记录上次的补全方式。

- `g:SuperTabDefaultCompletionType` 的值设置缺省的补全方式，缺省为 CTRL-P。

我们可以在 `~/.vimrc` 配置文件中设置这两个变量，例如：

```
let g:SuperTabRetainCompletionType = 2
let g:SuperTabDefaultCompletionType = "<C-X><C-O>"
```

其实就相当于将 `<C-X><C-O>` 映射到 Tab 键，但并不影响 Tab 键对齐的使用。现在我们可以使用 TAB 来进行补全了，新建一个 `test.c` 测试一下：

输入 `pri`，然后按 Tab 键，就会弹出自动补全下拉菜单，CTRL-N 移动光标到 `printf` 标签，CTRL-Y 选择并关闭下拉菜单。因为在前面我们已经将生成的包含所有系统头文件的标签文件添加到了 `~/.vimrc` 配置文件中，所以能够直接使用自动补全。

6、文件浏览和缓冲区浏览

1、目录浏览

我们在 vim 中执行命令：

```
:e ~/
```

或者在 shell 中执行：

```
Vim ~/
```

则显示当前用户跟目录的文件和子目录，因为当你使用 vim 尝试打开目录时，vim 会自动调用 `netrw.vim` 插件打开该目录因为执行该命令，而 `netrw.vim` 是 vim 的标准插件，它已经伴随 vim 而发行，不需要安装。

```
" =====
" Netrw Directory Listing                               (netrw v143)
" /home/topeet/ctags-5.8
" Sorted by      name
" Sort sequence: [/]\$. \<core\%(\\.d\\+\\)\$=\>.\\.h$. \.c$. \.cpp$. \~\$.*$. \.o$. \.obj$. \.info$. \.swp$. \.bak$.
" Quick Help: <F1>:help  -:go up dir  D:delete  R:rename  S:sort-by  X:exec
" =====
./
gnu_regex/
args.h
config.h
ctags.h
debug.h
e_amiga.h
e_djgpp.h
e_mac.h
e_msoft.h
e_os2.h
e_qdos.h
e_riscos.h
e_vms.h
entry.h
general.h
get.h
keyword.h
"~/ctags-5.8" is a directory                                     8.1      Top
```

netrw 支持本地系统和远端机器上的目录浏览；浏览包括列出文件和目录、进入目录、编辑那里的文件、删除文件/目录、建立新目录和移动（换名）文件和目录，复制文件和目录，等等。具体的功能实现，在 netrw 窗口按 F1 查看。

上面我们用“:e ~/”的方式打开 netrw，我们还可以使用“:Explore”等 Ex 命令来打开文件浏览器。通过阅读“:help netrw”帮助文档，我的 vimrc 中这样配置：

```
""""""""""
" netrw setting
""""""""""

let g:netrw_winsize = 15
let g:netrw_liststyle= 3 " 设置 netrw 列表风格为树状
let g:netrw_alto = 1 " 在 netrw 窗口下方打开所选文件
let g:netrw_altv = 1 " 在 netrw 窗口右侧打开所选文件
let g:netrw_list_hide= '\..*$, \.o$, \.out$ ' " 隐藏. 开头的文件以及.o
和.out 结尾的可执行文件

let g:netrw_browse_split = 3 " ,ex 快捷键打开时，选择文件，按回车键在
新标签页打开

nmap <silent> <leader>ex :Vexplore!<cr>
```

“:Sexplore[!] [dir]”表示分割并探索目录，这样，在我输入“， ex ”时，

就会打开一个垂直分隔的窗口浏览当前文件所在的目录，窗口的宽度为 15，位于右侧。

Netrw 插件中常用键绑定有：

```
<F1> netrw 给出帮助
-      netrw 往上一层目录
C      切换 vim 的当前工作目录为正在浏览的目录
d      创建目录
D      删除文件或目录
R      改名文件或目录
s      选择排序方式
x      定制浏览方式，使用你指定的程序打开该文件
o      用水平分割在新浏览窗口中进入光标所在的文件/目录
v      用垂直分割在新浏览窗口中进入光标所在的文件/目录
t      在新标签页里进入光标所在的文件/目录
```

如果在新标签页打开文件，可以通过 `gt` 组合键进行标签切换。

2、缓冲区浏览

在开发过程中，经常会打开很多缓冲区，尤其是使用 `tag` 文件在不同函数间跳转时，会不知不觉打开很多文件。要知道自己当前打开了哪些缓冲区，可以使用 `vim` 的 `:ls` 命令查看。

开发过程中，又经常需要在不同文件间跳转。我习惯于使用 `CTRL-^` 来切换文件，这就需要知道文件所在的缓冲区编号。每次都使用 `:ls` 来找缓冲区编号很麻烦，所以我使用 `BufExplorer` 插件来显示缓冲区的信息。

`BufExplorer` 插件在此处下载：

http://vim.sourceforge.net/scripts/script.php?script_id=42

下载后，把该文件在 `~/.vim/` 目录中解压缩，添加帮助文档：

```
toopeet@ubuntu:~/.vim$ unzip bufexplorer-7.4.6.zip
:helptags ~/.vim/doc
```

`BufExplorer` 已经映射了几个键绑定，由于在前面我们已经将 `mapleader` 设置为 `,"`，因此，使用 `,"bv` 就可以打开一个垂直分割窗口显示当前的缓冲区。

,be 或者 :BufExplorer 可以在当前窗口显示当前的缓冲区
,bs 或者 : BufExplorerHorizontalSplit 可以在水平分屏显示当前的缓冲区
,bv 或者 : BufExplorerVerticalSplit 可以在垂直分屏显示当前的缓冲区

在 BufExplorer 窗口按 F1 打开帮助菜单:

<shift-enter>或者 t : 新标签打开在缓冲区选中的文件
<enter>: 在当前缓冲区打开选中的文件

通过阅读 ":help BufExplorer" 帮助文档, 我的 vimrc 里这样设置 BufExplorer 插件:

```
""""""""""  
" BufExplorer  
""""""""""  
  
let g:bufExplorerDefaultHelp=0 " Do not show default help  
let g:bufExplorerShowRelativePath=1 " Show relative paths  
let g:bufExplorerSplitRight=0 " Split left  
let g:bufExplorerSplitVertical=1 " Split vertically  
let g:bufExplorerSplitVertSize = 30 " New split window is 30 columns wide  
let g:bufExplorerUseCurrentWindow=1 " Open in new window
```

3、winmanager 插件

winmanager 插件可以把前面介绍的 netrw 插件和 BufExplorer 插件集成在一起。

Winmanager 插件在这里下载:

http://vim.sourceforge.net/scripts/script.php?script_id=95

下载后, 把该文件在 ~/.vim/ 目录中解压缩:

```
topeet@ubuntu:~/.vim$ unzip winmanager.zip
```

仍然是用 ":helptags ~/.vim/doc" 命令来生成帮助标签, 然后就可以使用 ":help winmanager" 来查看帮助了。

使用 winmanager 插件可以控制各插件在 vim 窗口中的布局显示, 通过阅读 ":help winmanager" 帮助文档, 我的 vimrc 中这样设置:

```

" winManager setting
"

let g:winManagerWindowLayout = "FileExplorer,TagList|BufExplorer"
let g:winManagerWidth = 30
let g:bufExplorerMaxHeight = 10 " 缓冲区最大行数
let g:defaultExplorer = 0
nmap <silent> <leader>wm :WMToggle<cr>

```

`g:winManagerWindowLayout` 变量的值定义 winmanager 的窗口布局，插件间用“，”隔开，表示占用同一显示区域，通过 CTRL-N 切换在共享区域显示其中的插件；“|”隔开，则表示分屏显示，一旦启动 winmanager，会将整个 vim 分成左右两大部分，左屏称为“explorer area”，右屏称为“file editing area”，通过以上的配置，我们将 winmanager 打开/关闭的切换映射为“，wm”。打开 vim，在普通模式按“，wm”，效果如图：

```

56 flex.c
57 fortran.c
58 general.h
59 get.c
60 get.h
61 html.c
62 jscript.c
63 keyword.c
64 keyword.h
65 lisp.c
66 lregex.c
67 lua.c
68 mac.c
69 magic.diff
70 main.c
[File List] 70,1 34%
1 17 a main.c . l
2 16 lregex.c . l
3 15 keyword.c . l
4 14 get.c . l
5 13 flex.c . l
6 1 eiffel.c . l
7 12 dosbatch.c . l
8 11 debug.c . l
[Buf List] 1,2 Top ~/ctags-5.8/main.c

514 timeStamp (2);
515
516 if (Option.printTotals)
517     printTotals (timeStamps);
518 #undef timeStamp
519 }
520
521 /*
522 *      Start up code
523 */
524
525 extern int main (int unused argc, char **argv)
526 {
527     cookedArgs *args;
528 #ifdef VMS
529     extern int getredirection (int *ac, char ***av);
530
531     /* do wildcard expansion and I/O redirection */
532     getredirection (&argc, &argv);
533 #endif
534
535 #ifdef AMIGA
536     /* This program doesn't work when started from the Workbench */
537     if (argc == 0)

```

7、使用 lookupfile 插件

我们可以通过“:find”命令打开指定的文件，不过使用“:find”命令并不是非常的方便：一是如果项目比较大、文件比较多，find 查找起来很慢；二是必须输入全部的文件名，不能使用正则表达式(regex)查找。

我们也介绍过 vim 提供的文件浏览插件, 你可以在浏览器中根据目录去查找, 但这种方式在浏览目录时比较方便, 查找一个已知名字 (或知道部分名字) 的文件效率就比较低了。

Lookupfile 插件可以在下面的链接下载:

http://www.vim.org/scripts/script.php?script_id=1581

Lookupfile 插件需要最新的 genutils 支持, genutils 下载链接:

http://www.vim.org/scripts/script.php?script_id=197

下载后解压, 并且更新帮助文档。

1、项目文件查找

Lookupfile 在查找文件时, 需要使用 tag 文件。它可以使用 ctags 命令生成的 tag 文件, 不过查找效率会比较低。因此我们会专门为它生成一个包含项目中所有文件名的 tag 文件, 根据帮助文档 ":help lookupfile" 的说明, 编写一个 shell 脚本, 生成一个文件名 tag 文件。

以 ctags 源码为例, 切换到 ctags-5.8 目录, 步骤如下:

1、新建一个脚本文件:

```
topeet@ubuntu:~/ctags-5.8$ vi filenames.tags.sh
```

2、添加一下内容:

```
#!/bin/sh

echo -e "!_TAG_FILE_SORTED\t2\t/2=foldcase/" > filenames.tags

find . -not -regex '.*\.(jar|gif|jpg|class|pdd|sw[op]|xls|doc|pdf|zip|tar|ico|ear|war|dat|.*)' -type f -printf "%f\t%p\t1\n" | \

    sort -f >> filenames.tags
```

3、修改脚本文件的权限:

```
topeet@ubuntu:~/ctags-5.8$ chmod 777 filenames.tags.sh
```

4、执行脚本:

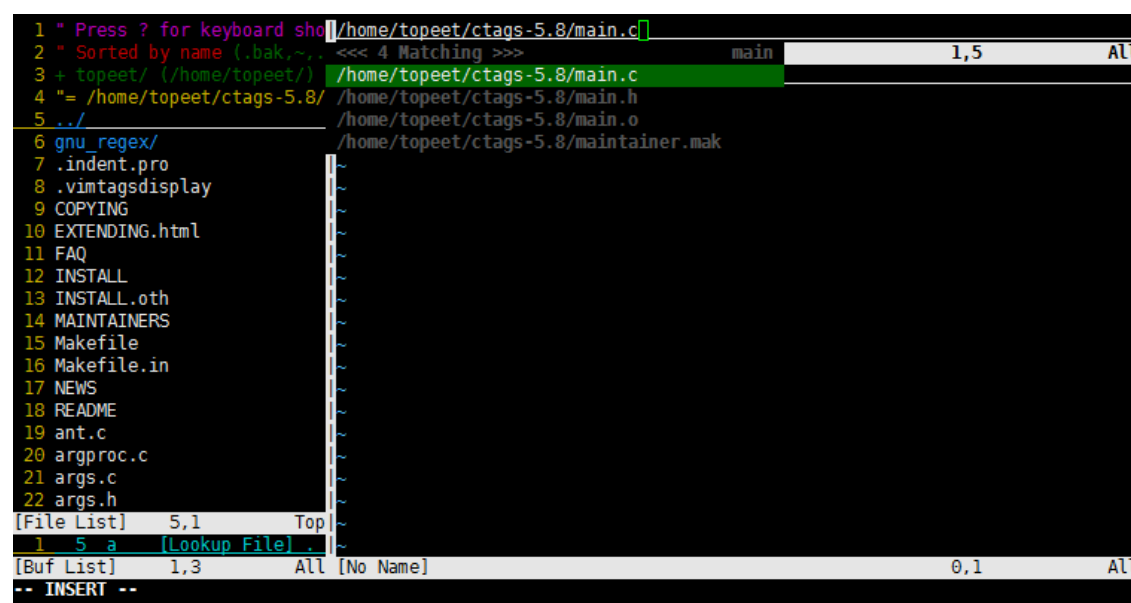
```
topeet@ubuntu:~/ctags-5.8$ ./filenames.tags.sh
```

5、指定 lookupfile 插件到哪去找文件名 tag 文件:

```
let g:LookupFile_TagExpr = string('./filenames.tags')
```

注：如果不设定 `g:LookupFile_TagExpr` 的值，那么 `lookupfile` 插件会以 `tags` 选项定义的文件作为它的 `tag` 文件。

现在我们就可以使用 `lookupfile` 来打开文件了，按“<F5>”或输入“:LookupFile”在当前窗口上方打开一个 `lookupfile` 小窗口，开始输入文件名（至少 4 个字符），随着你的输入，符合条件的文件就列在下拉列表中，这大大方便了文件的查找。我们可以用“CTRL-N”和“CTRL-P”来在下拉列表中选择你所需的文件。选中文件后，按回车，就可以在之前的窗口中打开此文件。



```
1 " Press ? for keyboard shortcuts
2 " Sorted by name (.bak,~,.) <<< 4 Matching >>>
3 + topeet/ (/home/topeet/) /home/topeet/ctags-5.8/main.c
4 "= /home/topeet/ctags-5.8/ /home/topeet/ctags-5.8/main.h
5 ../ /home/topeet/ctags-5.8/main.o
6 gnu_regex/ /home/topeet/ctags-5.8/maintainer.mak
7 .indent.pro
8 .vimtagsdisplay
9 COPYING
10 EXTENDING.html
11 FAQ
12 INSTALL
13 INSTALL.oth
14 MAINTAINERS
15 Makefile
16 Makefile.in
17 NEWS
18 README
19 ant.c
20 argproc.c
21 args.c
22 args.h
[File List] 5,1 Top
1 5 a [Lookup File] .
[Buf List] 1,3 All [No Name] 0,1 All
-- INSERT --
```

2、缓冲区查找

我们在之前提过 `BufExplorer` 插件，利用 `BufExplorer` 可以快速查找缓冲区的内容，但是随着我们缓冲区的内容不断增加，这种方式自然不显得不便。`Lookupfile` 插件提供了一个按缓冲区名字查找缓冲区的方式，使用“:LUBufs”命令开始在缓冲区中查找，它会根据你的输入，自动匹配指定缓冲区的名字，符合条件的缓冲区就显示在下拉列表中，选中所需缓冲区后，按回车，就会切换你所选的缓冲区。

进入 `vim`，输入命令“:LUBufs”进入缓冲区搜索，按两次退格键清空，再输入缓冲区的文件名，选择自己要搜索的文件，然后回车即可。

```
4 "= /home/topeet/ctags-5.8/[co]
5 ../ <<< 2 Matching >>> co
6 gnu_regex/ /home/topeet/ctags-5.8/config.log /home/topeet/ctags-5.8/[co]nfig.log
7 .indent.pro /home/topeet/ctags-5.8/config.status /home/topeet/ctags-5.8/[co]nfig.status
8 .vimtagsdisplay 7 debug=false
9 ant.c 8 ac_cs_recheck=false
10 argproc.c 9 ac_cs_silent=false
11 args.c 10 SHELL=${CONFIG_SHELL-/bin/bash}
12 asm.c 11 ## ----- ##
13 asp.c 12 ## M4sh Initialization. ##
14 awk.c 13 ## ----- ##
15 basic.c 14
16 beta.c 15 # Be more Bourne compatible
17 c.c 16 DUALCASE=1; export DUALCASE # for MKS sh
18 cobol.c 17 if test -n "${ZSH_VERSION+set}" && (emulate sh) >/dev/null 2>&1; then
19 config.h.in 18 emulate sh
20 config.log 19 NULLCMD=:
21 config.status 20 # Zsh 3.x and 4.x performs word splitting on ${1+"$@"}, which
22 configure 21 # is contrary to our usage. Disable this feature.
23 configure.ac 22 alias -g '${1+"$@"}'='"$@"'
24 COPYING 23 setopt NO_GLOB_SUBST
25 ctags 24 else
26 ctags.l 25 case `(set -o) 2>/dev/null` in
27 ctags.html 26 *posix*) set -o posix ;;
28 ctags.spec 27 esac
[File List] 21,1 1% 28
1 9 a [Lookup File] . 29 fi
2 8 a config.status . 30
3 7 config.log . 31
4 6 config.h.in . 32
5 1 main.c . 33
[Buf List] 1,3 All ~/ctags-5.8/config.status
-- INSERT --
```

3、浏览目录

Lookupfile 插件还提供了目录浏览的功能，使用“:LUWalk”打开 lookupfile 窗口后，同样需要按两次退格键清空，再输入目录关键字，lookupfile 会在下拉列表中列出这个目录中的所有子目录及文件供选择，如果选择了目录，就会显示这个目录下的子目录和文件；如果选择了文件，就在 vim 中打开这个文件。


```
4 "= /home/topeet/ctags-5.8/[]gnu_regex/[]
5 ../<<< 1 Matching >>> gn
6 gnu_regex/ gnu_regex/ / ./[gnu_regex f it exists.
7 .indent.pro
8 .vimgtagsdisplay
9 ant.c
10 argproc.c
11 args.c
12 asm.c
13 asp.c
14 awk.c
15 basic.c
16 beta.c
17 c.c
18 cobol.c
19 config.h.in
20 config.log
21 config.status
22 configure
23 configure.ac
24 COPYING
25 ctags
26 ctags.l
27 ctags.html
28 ctags.spec
29 debug.c
[File List] 21,1 1%
1 8 a config.status .
2 7 config.log .
3 6 config.h.in .
4 1 main.c .
[Buf List] 1,3 All ~/ctags-5.8/config.status
-- INSERT --
```

列举符合搜索的目录

```
4 "= /home/topeet/ctags-5.8/[]gnu_regex/regcomp.c[]
5 ../<<< 7 Matching >>> gnu_regex/
6 gnu_regex/ README.txt README.txt it exists.
7 .indent.pro regcomp.c regcomp.c
8 .vimgtagsdisplay regex.c regex.c
9 ant.c regex.h regex.h
10 argproc.c regex_internal.c regex_internal.c
11 args.c regex_internal.h regex_internal.h
12 asm.c regexec.c regexec.c
13 asp.c
14 awk.c
15 basic.c
16 beta.c
17 c.c
18 cobol.c
19 config.h.in
20 config.log
21 config.status
22 configure
23 configure.ac
24 COPYING
25 ctags
26 ctags.l
27 ctags.html
28 ctags.spec
29 debug.c
[File List] 21,1 1%
1 8 a config.status .
2 7 config.log .
3 6 config.h.in .
4 1 main.c .
[Buf List] 1,3 All ~/ctags-5.8/config.status
-- INSERT --
```

回车进入所选目录，并且在下拉菜单列举该目录的所有文件，回车选择并打开

Lookupfile 插件提供了一些配置选项，通过调整这些配置选项，使它更符合你的工作习惯。根据帮助文档” :help lookupfile” 配置我的 vimrc 中关于 lookupfile 的设置：

```

"lookupfile setting
"

let g:LookupFile_MinPatLength = 2 "最少输入 2 个字符才开始查找
let g:LookupFile_PreserveLastPattern = 0 "每次查找都清空输入栏
let g:LookupFile_AlwaysAcceptFirst = 1 "回车打开第一个匹配项目
let g:LookupFile_AllowNewFiles = 0 "禁止为不存在的文件创建缓冲区
let g:LookupFile_SearchForBufsInTabs = 1
"不显示可执行文件

Let g:LookupFile_FileFilter =
'\.class$\\\.o$\\\.obj$\\\.exe$\\\.jar$\\\.zip$\\\.war$\\\.ear$'
let g:LookupFile_TagExpr = string('./filenametags')
"映射 LookupFile 为, lk, <Plug>用于一个内部映射, 它不与任何键的序列匹配
nmap <silent> <leader>lk <Plug>LookupFile<cr>
"映射 LUBufs 为, lb
nmap <silent> <leader>lb :LUBufs<cr>
"映射 LUWalk 为, lw
nmap <silent> <leader>lw :LUWalk<cr>
set ignorecase " 忽略大小写的查找

```

在用 lookupfile 插件查找文件时, 是区分文件名的大小写的。因为 lookupfile 插件能够识别 'ignorecase' 和 'smartcase', 所以只要将 'ignorecase' 打开, 就可以每次在查找文件时都忽略大小写查找了。

8、集成编译 GCC 和调试 GDB

1、编译

使用过 Visual Studio 2013 编程的人都知道, 当我们编写完代码, 按 F7 键进行编译, 编译错误的信息显示在信息窗口, 双击可以跳转到错误行。利用 vim 的 quickfix 模式, 可以模仿这样的效果。

以 "test.c" 程序为例, 介绍一下 quickfix 模式的用法。

该程序的内容如下，里面包含了三个小小的错误：

```
1 #include <stdio.h>
2 int main()
3 {
4     int a = 10
5     int b = 20;
6     int c;
7     c = add(a, b);
8     printf("%d\n", c);
9     return 0;
10 }
```

我们可以为这个程序写个小小的 Makefile 文件，但是为了演示 'makeprg' 的设置方法，我们不用 Makefile，而直接设置 'makeprg' 选项，如下：

```
:set makeprg=gcc\ -Wall\ -g\ test.c\ -o\ test
```

执行以上命令会把 test.c 编译为名 test 的可执行文件，并打开了所有的 Warning 选项。如果编译命令中有空格，需要使用 '\' 对空格进行转义，上面的例子使用了 '\' 转义空格。

设置好 'makeprg' 选项后，输入下面的命令就可以编译了：

```
:make
```

在使用 ":make" 时，vim 会自动调用 'makeprg' 选项定义的命令进行编译，并把编译输出重定向到一个临时文件中，当编译出现错误时，vim 会从上述临时文件中读出错误信息，根据这些信息形成 quickfix 列表，并跳转到第一个错误出现的地方。

vim 会提示出错信息，如果你没看清出错信息，可以使用 ":cw" 命令，打开一个 quickfix 窗口，把所有的出错信息显示出来，效果如图：

```
1 #include <stdio.h>
2 int main()
3 {
4     int a = 10
5     int b = 20;
6     int c;
7     c = add(a, b);
8     printf("%d\n", c);
9     return 0;
10 }
11
~
~
test.c 1,5 All
1 || gcc test.c -o test
2 || test.c: In function 'main' :
3 test.c|5| error: expected ',' or ';' before 'int'
4 test.c|7| error: 'b' undeclared (first use in this function)
5 test.c|7| error: (Each undeclared identifier is reported only once
6 test.c|7| error: for each function it appears in.)
7 || make: *** [test] Error 1
~
~
~
[Quickfix List] :make
```

修改第一个，然后使用“:cn”命令跳到下一个错误，以此类推，直到修正全部错误。然后执行“:w”保存，再执行“:make”编译。

在 quickfix 模式里经常用到的命令有：

```
:cc 显示详细错误信息 ( :help :cc )
:cp 跳到上一个错误 ( :help :cp )
:cn 跳到下一个错误 ( :help :cn )
:cl 列出所有错误 ( :help :cl )
:cw 如果有错误列表，则打开 quickfix 窗口 ( :help :cw )
:col 到前一个旧的错误列表 ( :help :col )
:cnew 到后一个较新的错误列表 ( :help :cnew )
```

为了方便，我将常用的命令映射为快捷键，我的 vimrc 中 QuickFix 的配置：

```
""""""""""
" QuickFix setting
""""""""""

" 按下 F6，执行 make clean
map <F6> :make clean<CR><CR><CR>

" 按下 F7，执行 make 编译程序，并打开 quickfix 窗口，显示编译信息
autocmd FileType c,cpp map <F7> :make<CR><CR><CR> :cw10<CR><CR>
```

”使用”,cn”跳到下一个错误

```
nmap <leader>cn :cn<cr>
```

”使用”,cp”跳到上一个错误

```
nmap <leader>cp :cp<cr>
```

2、GDB 调试

1、下载最新的 vim74 的源码包:

<ftp://ftp.vim.org/pub/vim/unix/vim-7.4.tar.bz2>

2、下载 vimgdb-for-vim7.4 源码:

<https://github.com/larrupingpig/vimgdb-for-vim7.4/archive/master.zip>

3、打补丁:

下载完 vim 源码和 vimgdb 补丁,我是在用户根目录新建一个 vim 目录,然后将它们剪切到该目录,并且解压和打补丁:

```
topeet@ubuntu:~/vim$ tar xjvf vim-7.4.tar.bz2
topeet@ubuntu:~/vim$ unzip vimgdb-for-vim7.4-master.zip
topeet@ubuntu:~/vim$ patch -p0 < vimgdb-for-vim7.4-master/vim74.patch
topeet@ubuntu:~/vim$ cd vim74/src
topeet@ubuntu:~/vim$ make
topeet@ubuntu:~/vim$ make install
```

注意:编译和安装 vim 只使用下面两个命令:

```
make
make install
```

而没有执行./configure,如果你要指定安装路径,需要执行./configure的话,必须要加上:--enable-gdb

我在 make 的时候,出现了一个错误,切换到 auto 目录查看 config.log 文件,搜索 error:

```
conftest.c:10: fatal error: ac_nonexistent.h: No such file or directory
```

在网上搜索大半天没解决,可能是编译 gvim 不成功,我先安装一些依赖库(这个步骤可省略),然后直接禁止 GUI 就可以了。

```
topeet@ubuntu:~/C_code/test$ sudo apt-get update
```

```
topeet@ubuntu:~/C_code/test$ sudo apt-get install libncurses5-dev
libgnome2-dev      libgnomeui-dev      libgtk2.0-dev      libatk1.0-dev
libbonoboui2-dev  libcairo2-dev      libx11-dev         libxpm-dev         libxt-dev
python-dev ruby-dev mercurial
topeet@ubuntu:~/vim/vim74/src$ ./configure --enable-gdb --disable-gui
topeet@ubuntu:~/vim/vim74/src$ make
topeet@ubuntu:~/vim/vim74/src$ sudo make install
```

由于我没有指定安装路径，安装完毕以后 vim 放在了 vim 源码的 src 目录下了，为了让所有用户在任意地方使用，我直接将它覆盖原版本的 vim:

```
topeet@ubuntu:~/C_code/test$ sudo vim /usr/bin/
topeet@ubuntu:~/C_code/test$ vim --version
VIM - Vi IMproved 7.4 (2013 Aug 10, compiled Apr 9 2016 05:39:51)
Compiled by topeet@ubuntu
```

安装 vimGdb 运行文件:

如果你不确定你具体目录，可以在打开的 vim 中执行一下命令查看环境中的具体目录:

```
:set runtimepath?
topeet@ubuntu:~/.vim$
cp -rf ~/vim/vimgdb-for-vim7.4-master/vimgdb_runtime/* ~/.vim
```

然后启动 vim，更新帮助文档:

```
:helptags ~/.vim/doc
```

至此，我们可以使用命令“:help vimgdb”查看帮助文档了。

[在 vim 中调试]

首先确保你的计算机上安装了 gdb，vimgdb 支持 5.3 以上的 gdb 版本。我们使用下面这个简单的例子，来示例一下如何在 vim 中使用 gdb 调试。看示例代码:

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```

3 int main(int argc, char **argv)
4 {
5     int i = 0;
6     int array[5];
7     for (i = 0; i < 5; i++)
8     {
9         array[i] = i + 1;
10        printf("%d ", array[i]);
11    }
12    printf("\n");
13    exit(0);
14    return 0;
15 }

```

文件/home/topeet/C_code/test/src/test.c 内容如上，在该目录下编写一个 Makefile 文件，执行”，ex” 打开文件浏览（我们在前面映射文件浏览的快捷键），按 shift-%组合键新建文件，输入文件名为 Makefile，回车会在本窗口打开我们新建的文件，输入以下代码：

```

1 .PHONY = all, clean
2 CC = gcc
3 OBJS = test.c
4 PARAM =-g -w
5
6 all:$(OBJS)
7     $(CC) $(PARAM) $< -o $@
8
9 clean:
10     rm -r *.o

```

保存退出，切换到 test.c 窗口，执行” :w” 保存文件（不退出），然后按 F7 进行编译，没有弹出错误信息 quickfix 窗口说明没有语法错误，生成了最终的

test 文件,我们就可以进行调试了。但在第一次调试之前,需要修改一下 vimgdb 补丁绑定的键:

```
topeet@ubuntu:~/C_code/test/src$ cd ~/.vim/macros/  
topeet@ubuntu:~/vim/macros$ vi gdb_mappings.vim
```

修改 29、30 行的键值映射,因为我们在前面映射 F7 为编译了,根据 Visual Studio 2013 风格,我把它修改为 F5,也就是 F5 键为缺省定义和调试命令间切换。

vimgdb 补丁已经定义了一些键绑定,使用前我们先加载这些绑定:

```
:run macros/gdb_mappings.vim
```

为了使用方便,直接在 ~/.vimrc 配置文件添加:

```
run macros/gdb_mappings.vim
```

加载后,按<F5>可以在按键的缺省定义和调试命令间切换。在底部会显示”gdb keys mapped”,再按空格键,会在当前窗口下方会打开一个小窗口(command-line 窗口),这就是 vimgdb 的命令窗口,可以在这个窗口中输入任何合法的 gdb 命令,输入的命令将被送到 gdb 执行。

接下来,在 command-line 窗口中输入以下命令:

```
file test
```

这条命令加载我们编译的 test 程序准备调试。现在使用 vim 的移动命令,把光标移动到 test.c 的第 5 行和 12 行,按”CTRL-B”在这两处设置断点,现在 vim 看起来是这样:

```
 7  There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
 8  and "show warranty" for details.  
 9  This GDB was configured as "x86_64-linux-gnu".  
10  For bug reporting instructions, please see:  
11  <http://bugs.launchpad.net/gdb-linaro/>.  
12 (gdb) file test  
13 Reading symbols from /home/topeet/C_code/test/src/test...done.  
14 (gdb) break "/home/topeet/C_code/test/src/test.c:5"  
15 Breakpoint 1 at 0x400593: file test.c, line 5.  
16 (gdb) break "/home/topeet/C_code/test/src/test.c:12"  
17 Breakpoint 2 at 0x4005d9: file test.c, line 12.  
18 (gdb)  
/tmp/vQuB9H5/2[-] [POS=0018,0001] [100%] [LEN=18] [FORMAT=unix] [TYPE=] [ASCII=040] [HEX=28]  
1  5  int i = 0;  
   6  int array[5];  
   7  for (i = 0; i < 5; i++)  
   8  {  
   9      array[i] = i + 1;  
  10      printf("%d ", array[i]);  
  11  }  
2 12  printf("\n");  
   13  exit(0);  
   14  return 0;  
   15 }  
~/C_code/test/src/test.c [POS=0012,0005] [80%] [LEN=15] [FORMAT=unix] [TYPE=C] [ASCII=009] [HEX=09]
```


断点所在的行以蓝色为背景，并在行前显示标记 1 和 2 表明是第几个断点；调试窗口也显示入栈信息。

然后按“R”，使 gdb 开始运行，停在我们设置的第一个断点处，如图所示：

```
11 <http://bugs.launchpad.net/gdb-linaro/>.
12 (gdb) file test
13 Reading symbols from /home/topeet/C_code/test/src/test...done.
14 (gdb) break "/home/topeet/C_code/test/src/test.c:5"
15 Breakpoint 1 at 0x400593: file test.c, line 5.
16 (gdb) break "/home/topeet/C_code/test/src/test.c:12"
17 Breakpoint 2 at 0x4005d9: file test.c, line 12.
18 (gdb) run
19 Starting program: /home/topeet/C_code/test/src/test
20
21 Breakpoint 1, main (argc=1, argv=0x7fffffffe5f8) at test.c:5
22 (gdb)
/tmp/vQuB9H5/2[-] [POS=0022,0001] [100%] [LEN=22] [FORMAT=unix] [TYPE=] [ASCII=040] [HEX=28]
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char **argv)
4 {
5 => int i = 0;
6 int array[5];
7 for (i = 0; i < 5; i++)
8 {
9     array[i] = i + 1;
10    printf("%d ", array[i]);
11 }
```

程序当前运行到的行以黄色为背景，行前以“=>”指示，表明这是程序执行的位置。

接下来，我们再按“CTRL-N”单步调试，效果如图：

```
17 Breakpoint 2 at 0x4005d9: file test.c, line 12.
18 (gdb) run
19 Starting program: /home/topeet/C_code/test/src/test
20
21 Breakpoint 1, main (argc=1, argv=0x7fffffffe5f8) at test.c:5
22 (gdb) next
23 (gdb)
~
~
~
~
~
/tmp/vQuB9H5/2[-] [POS=0023,0001] [100%] [LEN=23] [FORMAT=unix] [TYPE=] [ASCII=040] [HEX=28]
2 #include <stdlib.h>
3 int main(int argc, char **argv)
4 {
5 int i = 0;
6 int array[5];
7 => for (i = 0; i < 5; i++)
8 {
9     array[i] = i + 1;
10    printf("%d ", array[i]);
11 }
2 12 printf("\n");
~/C_code/test/src/test.c [POS=0007,0004] [46%] [LEN=15] [FORMAT=unix] [TYPE=C] [ASCII=009] [HEX=09]
```

按“C”，运行到第 2 个断点处，现在，我们输入下面的 vim 命令，在右下方分隔出一个名为 gdb-variables 的宽度为 20 的窗口：

```
:below 20vsplit gdb-variables
```

然后用“v”命令选中变量 i，按“CTRL-P”命令，把变量 i 加入到监视窗口，用“5v”命令选中变量 array，按“CTRL-P”命令，同样把变量 array 加入到监视窗口，同样，也可以按空格弹出 command-line 窗口，输入 createvar array(createvar 后面跟变量名)，将变量加入到监视窗口，效果如图：

```

23 (gdb) break "/home/topeet/.vim/doc/gdb.txt:274"
24 No source file named /home/topeet/.vim/doc/gdb.txt.
25 Make breakpoint pending on future shared library load? (y or [n]) n
26 (gdb)
27 (gdb) continue
28 Continuing.
29
30 Breakpoint 2, main (argc=1, argv=0x7fffffffe5f8) at test.c:12
31 (gdb)
32 (gdb)
33 (gdb)

/tmp/vQuB9H5/2[-] [POS=0033,0001] [100%] [LEN=33] [FORMAT=unix] [TYPE=] [ASCII=040] [HEX=28]
1 4 {
2 5     int i = 0;
3 6     int array[5];
4 7     for (i = 0; i < 5; i++)
5 8     {
6 9         array[i] = i + 1;
7 10        printf("%d ", array[i]);
8 11    }
9 => 12    printf("\n");
10 13    exit(0);
11 14    return 0;
12 15 }

</src/test.c [POS=0010,0023] [66%] [LEN=15] [FORMAT=unix] [TYPE=C] [ASCII=097] [HEX=61] <ASCII=051] [HEX=33]

@ 26 file all
@ 27 createvar i
@ 28
<nd Line] [POS=0028,0001] [100%] [LEN=28] [FORMAT=unix] [TYPE=GDB] [ASCII=000] [HEX=00]

```

```

2 #include <stdlib.h>
3 int main(int argc, char **argv)
4 {
1 5     int i = 0;
2 6     int array[5];
3 => 7     for (i = 0; i < 5; i++)
4 8     {
5 9         array[i] = i + 1;
6 10        printf("%d ", array[i]);
7 11    }
8 12    printf("\n");
9 2

</src/test.c [POS=0007,0001] [46%] [LEN=15] [FORMAT=unix] [TYPE=C] [ASCII=009] [HEX=09] <ASCII=049] [HEX=31]

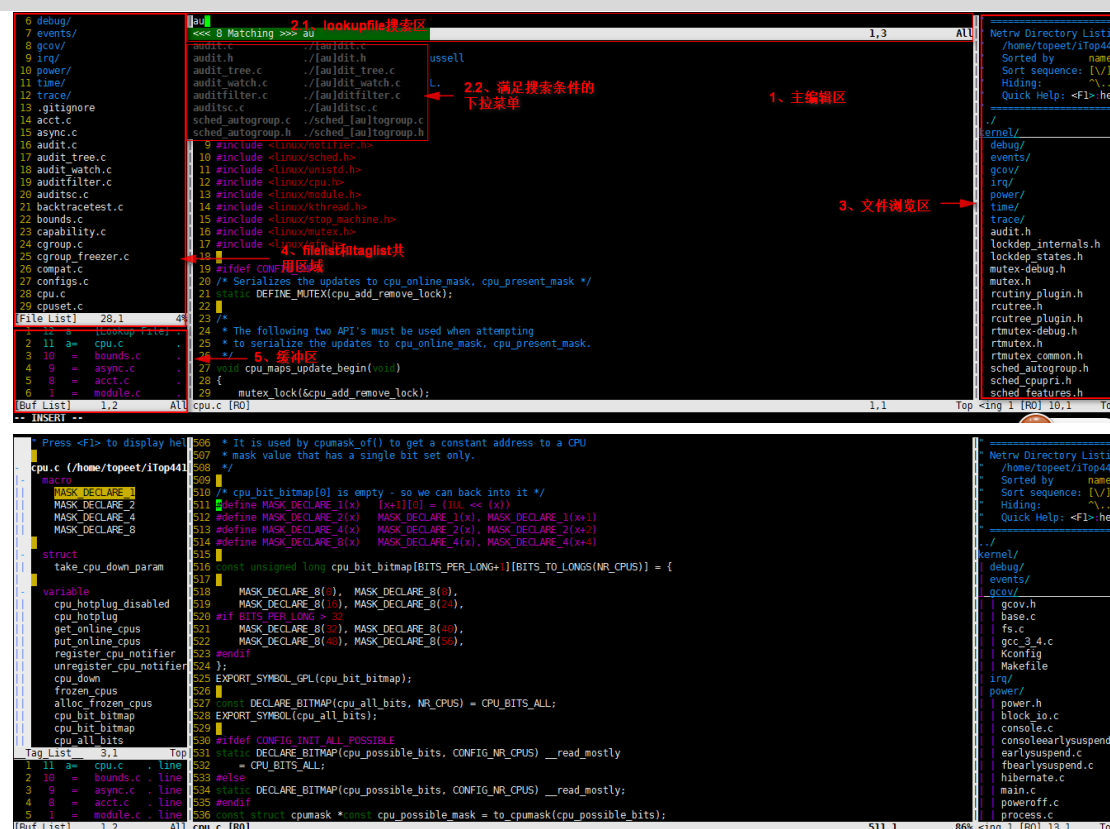
```

注意要在 gdb-variables 窗口在按空格，如果光标在主编辑窗口的话，输入 createvar i 回车以后，会覆盖主编辑窗口，这样就无法调试了。

以上使用的快捷键，都是 gdb_mappings.vim 定义的键绑定，常用的键说明：

:	空格	打开 command-line 窗口
B		打印断点信息
A		参数信息
S		逐步调试（进入函数）
CTRL-N		单步调试

F	结束，返回堆栈信息，但是不退出
R	运行
Q	退出
C	continue
W	where
CTRL-B	设置断点
CTRL-E	取消光标所在行的断点
CTRL-P	Normal mode: 在 Gdb 窗口打印光标下变量的值 Visual mode: 把变量 i 加入到监视窗口 (gdb-variables)
CTRL-X	打印变量所在的内存地址



总结

经过以上的配置，涉及的快捷键如下：

- ,wm 打开 winmanager 插件
- ,ee 打开文件浏览
- ,bv 垂直分屏打开缓冲区

,bs 水平分屏打开缓冲区
,bo 打开缓冲区
,lb 打开缓冲区搜索
,lw 打开路径搜索
,lk 打开搜索(需要先生成 tags)
<F3> 快速生成 tags 标签
<F5> 进入调试模式
<F7> 编译(当前目录需要存在 Makefile 文件)
CTRL-L 行补全
CTRL-F 文件补全
CTRL-D 字典补全
CTRL-Y 选择并关闭补全下拉菜单
CTRL-E 关闭补全下拉菜单, 并恢复补全前的状态
CTRL-N 往下选
CTRL-P 往上选

演示利用 vim 建立一个测试工程过程:

- 1、启动 vim
- 2、,ex 快捷键打开文件浏览
- 3、d 键创建工程目录(test)
- 4、回车进入工程目录, d 键创建子目录(src、include)
- 5、%键创建名为 Makefile 的文件
- 6、保存退出
- 7、重新打开文件浏览
- 8、同样的方式分别在 src、include 目录新建文件
- 9、光标停在所选的文件, t 键在新标签打开文件, 然后编辑