

手把手教你把 Vim 改装成一个 IDE 编程环境(图文)

标签: [vim 编程](#) [idesearchbuffertags](#)

2011-07-23 17:54 10785 人阅读 [评论](#) (0) [收藏](#) [举报](#)

分类:

Unix/Linux (14) ▾

By: 吴垠

Date: 2007-09-07

Version: 0.5

Email: lazy.fox.wu@gmail.com

Homepage: <http://blog.csdn.net/wooin>

Copyright: 该文章版权由吴垠和他可爱的老婆小包子所有。可在非商业目的下任意传播和复制。对于商业目的下对本文的任何行为需经作者同意。

联系方式: lazy.fox.wu@gmail.com

1 写在前面

Linux 下编程一直被诟病的一点是: 没有一个好用的 IDE, 但是听说 Linux 牛人, 黑客之类的也都不用 IDE. 但是对我等从 Windows 平台转移过来的 Coder 来说, 一个好用的 IDE 是何等的重要啊, 估计很多人就是卡在这个门槛上了, “工欲善其事, 必先利其器”嘛, 我想如果有一个很好用的 IDE, 那些 Linux 牛人也会欢迎的. 这都是劳动人民的美好愿望罢了, 我今天教大家把 gvim 改装成一个简易 IDE, 说它“简易”是界面上看起来“简易”, 但功能绝对不比一个好的 IDE 差, 该有的功能都有, 不该有的功能也有, 你就自己没事偷着乐吧, 下面我开始介绍今天的工作了.

本文会教你:

1. [中文帮助手册的安装](#)
2. [vim 编程常用命令](#)
3. [语法高亮](#)
4. [在程序中跳来跳去: Ctags 的使用](#)
5. [教你高效地浏览源码 -- 插件: TagList](#)

6. 文件浏览器和窗口管理器 -- 插件: WinManager
7. Cscope 的使用
8. QuickFix 窗口
9. 快速浏览和操作 Buffer -- 插件: MiniBufExplorer
10. c/h 文件间相互切换 -- 插件: A
11. 在工程中查找 -- 插件: Grep
12. 高亮的书签 -- 插件: VisualMark
13. 自动补全
14. 加速你的补全 -- 插件: SuperTab

本文不会教你:

1. 如何使用 vim. 本文不会从零开始教你如何使用 vim, 如果你是第一次接触 vim, 建议你先看看其他的 vim 入门的教程, 或者在 shell 下输入命令: vimtutor, 这是一个简单的入门教程.
2. 编程技巧.
3. vim 脚本的编写.

我的工作环境是: Fedora Core 5

gvim 是自己编译的 7.0, 如果你还没有安装 gvim, 请看我的这篇文章<在 Redhat Linux 9 中编译和配置 gvim 7.0>

由于本人一直从事 C 语言工作, 所以下面这些例子都是在 C 语言程序中演示的, 其他语言的没有试过, 如果有朋友在别的语言下有问题, 可以跟我讨论一些, 我会尽量帮助你们的.

本文用的示范源码是 vim7.1 的源码, 可以在 www.vim.org 下载到源码包: vim-7.1.tar.bz2, 你也可以不用下载, 就用你自己程序的源码, 关系不大的. 我把源码解压在我的 home 目录下: ~/vim71

下面对文中的一些名字定义一下:

1. 文中用到的一些用<>括起来的符号比如<C-T>, <C-S-A>, 之类的, 你可以用下面的命令看看解释:

```
:help keycodes
```

2. 文中说的一些路径, 比如:

`~/vim/plugin`

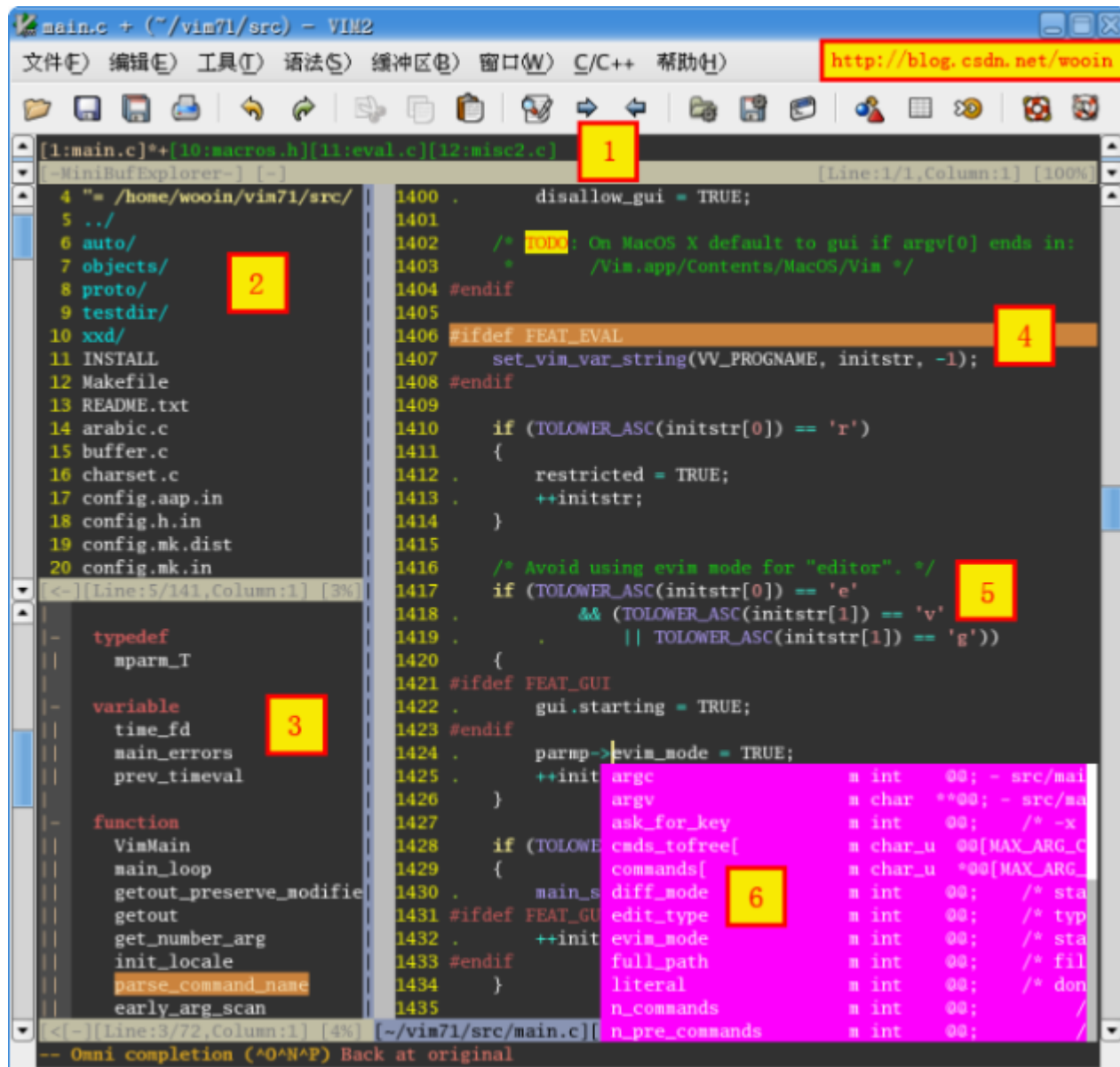
`~/vim/doc`

`~/vim/syntax`

如果你没有，就自己创建.

3. 文中说到的.vimrc 文件都是指 `~/vimrc`

先给大家看张图片，我是 vim 的界面，解解馋先^_^



(--- 图 1 ---)

对照上图的图标，我在本文中将教会你以下这些功能：

- 1 简洁明了的 Buffer 浏览和操作

2	文件浏览器
3	tag 浏览器
4	高亮的书签
5	更丰富的语法高亮
6	成员变量的下拉， 自动补全

2 中文帮助手册的安装

vim 自带的帮助手册是英文的，对平时编程的人来说没有多大阅读困难，何况还有“星级译王”呢，可偏偏有一帮人将其翻译成了中文，可偏偏我又挡不住诱惑将它安装了，唉..... 又痛失一个学习英文的好机会，下不为例。
大家看看我的中文帮助界面吧：



[1:main.c]*[2:misc2.c]

[-MiniBufExplorer-] [-]

[Line:1/1,Column:1] [100%]

help.txt For Vim version 7.0. 最近更新:2006年6月

VIM - 主帮助文件

移动: 使用光标键, 或者用 "h" 向左, "j" 向下, "k" 向上, "l" 向右。

关闭本窗口: 使用 ":q<Enter>"。

离开 Vim: 使用 ":qa!<Enter>" (当心, 所有的改动都会丢失!)。

跳转到一个主题: 将光标置于标签 (例如 |bars|) 上然后输入 CTRL-]。

使用鼠标: ":set mouse=a" 启动对鼠标的支持 (用于 xterm 或 GUI)。

在标签 (例如 |bars|) 上双击。

跳回: 键入 CTRL-T 或 CTRL-O (重复则继续向后)。

获取特定帮助: 在 |:help| 命令后给出参数可以直接跳转到任何的帮助主题。

还可以进一步指定上下文:

help-context

类别

前缀

例子 ~

[help.cnx][help] [RO][-]

[Line:3/214,Column:18] [1%]

2796 * Setup to start using the GUI. Exit with an error when not available.

2797 */

2798 static void

2799 main_start_gui()

2800 {

2801 #ifdef FEAT_GUI

2802 gui.starting = TRUE;. /* start GUI a bit later */

2803 #else

2804 mch_errmsg(_(e_nogvim));

2805 mch_errmsg("\n");

2806 mch_exit(2);

2807 #endif

2808 }

2809

2810 /*

2811 * Get an environment variable, and execute it as Ex commands.

2812 * Returns FAIL if the environment variable was not executed, OK otherwise.

2813 */

[~/vim71/src/main.c][c]

[Line:2809/3841,Column:0] [73%]

:help

(--- 图 2 ---)

安装方法:

在下面的网站下载中文帮助的文件包:

<http://vimdoc.sf.net> (English)

<http://vcd.gro.clinux.org> (中文)

下载的文件包应该是类似这样的: vimdoc-1.5.0.tar.gz

解压后其中有个 doc 文件夹, 将其中的内容全部复制到 ~/.vim/doc, 或者 vim 安装目录下的 doc 目录中, 此时 vim 中的 help 信息已经是中文的了.

注意:

a. 如果无法显示中文, 在 ~/.vimrc 中增加下面这句试试:

```
set helplang=cn
```

b. 帮助文件的文本是 utf-8 编码的, 如果想用 vim 直接查看, 需要在 ~/.vimrc 中设置:

```
set encoding=utf-8
```

3 vim 编程常用命令

建议先看看帮助手册中的下面章节, 其中有关 tags 文件的部分你可以先跳过, 在后面的章节中会讲到, 到时候你在回来看看, 就觉得很简单了:

```
:help usr_29
```

```
:help usr_30
```

下面是我常用的一些命令, 放在这里供我备忘:

%	跳转到配对的括号去
[[跳转到代码块的开头去(但要求代码块中'{'必须单独占一行)
gD	跳转到局部变量的定义处
''	跳转到光标上次停靠的地方, 是两个', 而不是一个"
mx	设置书签, x 只能是 a-z 的 26 个字母
`x	跳转到书签处("`"是 1 左边的键)
>	增加缩进, "x>"表示增加以下 x 行的缩进
<	减少缩进, "x<"表示减少以下 x 行的缩进

4 语法高亮

写程序没有语法高亮将是一件多么痛苦的事情啊，幸亏 vim 的作者是个程序员(如果不是，那可 NB 大了)，提供了语法高亮功能，在上面的图片中大家也可以看到那些注释，关键字，字符串等，都用不同颜色显示出来了，要做到这样，首先要在你的 `~/.vimrc` 文件中增加下面几句话：

```
syntax enable  
syntax on
```

再重新启动 vim，并打开一个 c 程序文件，是不是觉得眼前突然色彩缤纷了起来...

如果你不喜欢这个配色方案你可以在“编辑->配色方案”(gvim)中选择一个你满意的配色方案，然后在 `~/.vimrc` 文件中增加下面这句：

```
colorscheme desert
```

desert 是我喜欢的配色方案，你可以改成你的。如果菜单中的配色方案你还不满意(你也太花了吧)，没关系，在 `vim.org` 上跟你一样的人很多，他们做了各种各样的颜色主题，你可以下载下来一个一个的试，多地可以看到你眼花。如果这样你还不满意(你还真是 XXXX)，没关系，vim 的作者早想到会有你这种人了，你可以创建你自己的颜色主题，把下面的这篇文档好好学习一些一下吧：

```
:help syntax.txt
```

更炫的语法高亮：

你可能会发现很多东西没有高亮起来，比如运算符号，各种括号，函数名，自定义类型等，但是看上面的图片，我的运算符号和函数名都加亮了^_^，想知道为什么吗？哇哈哈哈哈哈... 让我来教你吧 ...

主要的思路是新建一个语法文件，在文件中定义你要高亮的东东，想高亮什么就高亮什么，用 vim 就是这么自信。所谓的语法文件就是 vim 用来高亮各种源文件的一个脚本，vim 靠这个脚本的描述来使文件中的不同文本显示不同的颜色，比如 C 语言的语法文件放在类似于这样的路径中：

```
/usr/share/vim/vim64/syntax/c.vim
```

其他语言的语法文件也可以在这个路径中找到，你的也许不在这个路径中，不管它，在你自己的 HOME 下新建一个语法文件，新建一个空文件：

```
~/.vim/syntax/c.vim
```

在其中加入


```

"=====
Highlight All Function
"=====
syn match cFunction "/<[a-zA-Z_][a-zA-Z_0-9]*/>[^()]*" ("me=e-2
syn match cFunction "/<[a-zA-Z_][a-zA-Z_0-9]*/>/s*("me=e-1
hi cFunction gui=NONE guifg=#B5A1FF
"=====

Highlight All Math Operator
"===== " C math
operators syn match cMathOperator display "[+/*/%=]" " C
pointer operators syn match cPointerOperator display "->/|/."
" C logical operators - boolean results
syn match cLogicalOperator display "[!<>]=/"
syn match cLogicalOperator display "==" " C bit operators
syn match cBinaryOperator display "/(&/|/|/~/|<</|>>/)=/"
syn match cBinaryOperator display "/~"
syn match cBinaryOperatorError display "/^=" " More C logical
operators - highlight in preference to binary
syn match cLogicalOperator display "&&/|/"
syn match cLogicalOperatorError display "/(&&/|/|/)=/" " Math
Operator hi cMathOperator guifg=#3EFFE2
hi cPointerOperator guifg=#3EFFE2
hi cLogicalOperator guifg=#3EFFE2
hi cBinaryOperator guifg=#3EFFE2
hi cBinaryOperatorError guifg=#3EFFE2
hi cLogicalOperator guifg=#3EFFE2
hi cLogicalOperatorError guifg=#3EFFE2

```

再打开你的C文件看看，是不是又明亮了许多。还有一个压箱底的要告诉你，如果你自己增加了一个类型或者结构之类的，怎么让它也象“int”，“void”这样高亮起来呢？再在上面的文件~/.vim/syntax/c.vim中添加下面的东东：

```

"=====

```

```
" My Own DataType
"=====
syn keyword cType My_Type_1 My_Type_2 My_Type_3
```

这样你自己的类型My_Type_1, My_Type_2, My_Type_3就也可以向"int"一样高亮起来了, 这样的缺点是每增加一个类型, 就要手动在这里添加一下, 如果有人知道更简单的方法请一定一定要告诉我, 用下面的地址:

Email	: lazy.fox.wu@gmail.com
Homepage	: http://blog.csdn.net/wooin

5 在程序中跳来跳去: Ctags 的使用

哇, 这下可厉害了, Tag 文件(标签文件)可是程序员的看家宝呀, 你可以不用它, 但你不能不知道它, 因为Linux内核源码都提供了"make tags"这个选项. 下面我们就来介绍 Tag 文件.

tags 文件是由 ctags 程序产生的一个索引文件, ctags 程序其是叫"Exuberant Ctags", 是 Unix 上面 ctags 程序的替代品, 并且比它功能强大, 是大多数 Linux 发行版上默认的 ctags 程序. 那么 tags 文件是做什么用的呢? 如果你在读程序时看了一个函数调用, 或者一个变量, 或者一个宏等等, 你想知道它们的定义在哪儿, 怎么办呢? 用 grep? 那会搜出很多不相干的地方. 现在流行用是的是<C-]>, 谁用谁知道呀, 当光标在某个函数或变量上时, 按下"Ctrl+]", 光标会自动跳转到其定义处, 够厉害吧, 你不用再羡慕 Visual Studio 的程序员了, 开始羡慕我吧~_~.

你现在先别急着去按<C-]>, 你按没用的, 要不要我干什么呀, 你现在要做的是查查你电脑里有没有 ctags 这个程序, 如果有, 是什么版本的, 如果是 Ctags 5.5.4, 就象我一样, 你最好去装一个 Ctags 5.6, 这个在后面的自动补全章节中会用到. 在这个网站:

<http://ctags.sourceforge.net>, 下载一个类似 ctags-5.6.tar.gz 的文件下来(现在好像 5.7 版的也出来了, 不过我还没用过):

用下面的命令解压安装:

```
$ tar -xzvf ctags-5.6.tar.gz
$ cd ctags-5.6
$ make
```

```
# make install // 需要 root 权限
```

然后去你的源码目录，如果你的源码是多层的目录，就去最上层的目录，在该目录下运行命令：ctags -R

我现在以 vim71 的源码目录做演示

```
$ cd /home/wooin/vim71  
$ ctags -R
```

此时在/home/wooin/vim71 目录下会生成一个 tags 文件，现在用 vim 打开 /home/wooin/vim71/src/main.c

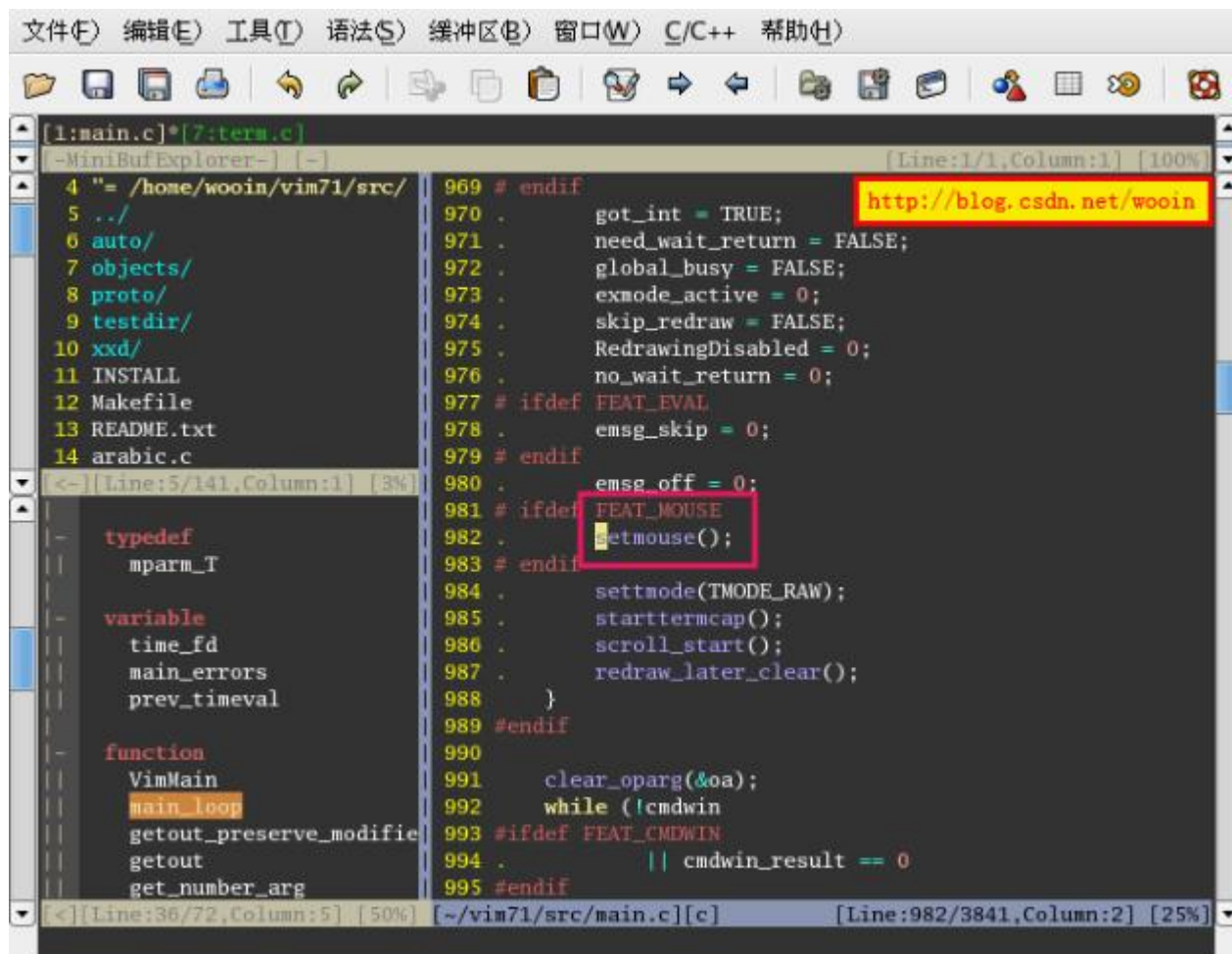
```
$ vim /home/wooin/vim71/src/main.c
```

再在 vim 中运行命令：

```
:set tags=/home/wooin/vim71/tags
```

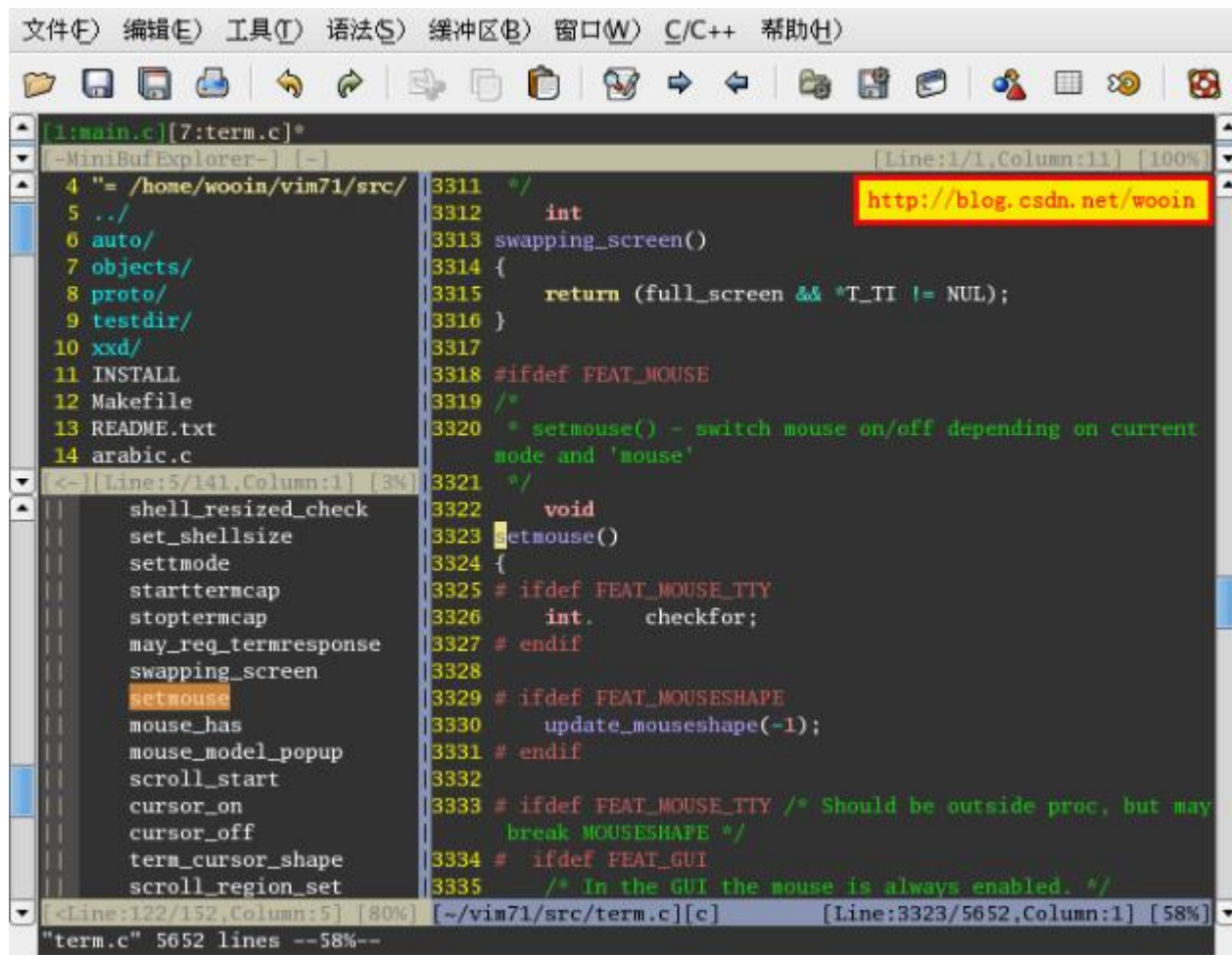
该命令将 tags 文件加入到 vim 中来，你也可以将这句话放到 ~/.vimrc 中去，如果你经常在这个工程编程的话。

下面要开始真刀实枪的开干了，如下图，将光标放在 setmouse() 函数上



(--- 图 3 ---)

此时按下<C-]>，光标会自动跳到 `setmouse()` 函数的定义处，见下图：



(--- 图 4 ---)

如果此时你还想再跳回刚才的位置，你还可以按<C-T>，这样又跳回到 setmouse() 函数被调用的地方了，变量，结构，宏，等等，都可以的，赶快试试吧.....

此时在回头学习一下第 3 节中说的 vim 手册吧

```
:help usr_29
```

不过还有一个小瑕疵，你修改程序后，比如增加了函数定义，删除了变量定义，tags 文件不能自动 rebuild，你必须手动再运行一下命令：

```
$ ctags -R
```

使 tags 文件更新一下，不过让人感到欣慰的是 vim 不用重新启动，正在编写的程序也不用退出，马上就可以又正确使用<C-]>和<C-T>了。如果有人知道更简单的方法请一定一定要告诉我，用下面的地址：

Email	: lazy.fox.wu#gmail.com
Homepage	: http://blog.csdn.net/wooin

6. 教你高效地浏览源码 -- 插件: TagList

下载地址	http://www.vim.org/scripts/script.php?script_id=273
版本	4.4
安装	在 ~/.vim 目录下解压 taglist_xx.zip
手册	:help taglist.txt

在 Windows 平台上用过 Source Insight 看程序的人肯定很熟悉代码窗口左边那个 Symbol 窗口，那里面列出了当前文件中的所有宏，全局变量，函数名等，在查看代码时用这个窗口总揽全局，切换位置相当方便，今天告诉你一个 vim 的插件: Taglist，可以同样实现这个功能。

上一节已经告诉你 ctags 的用法了，ctags 的基本原理是将程序程序中的一些关键字(比如：函数名，变量名等)的名字，位置等信息通过一个窗口告诉你，如果你已经安装好 taglist，则可以用下面的命令看看 taglist 自带的帮助文件：

```
:help taglist.txt
```

下面是我翻译的其中的第一段“Overview”，供大家现了解一下 taglist，翻译的不好，请指教：

“Tab List”是一个用来浏览源代码的 Vim 插件，这个插件可以让你高效地浏览各种不同语言编写的源代码，“Tag List”有以下一些特点：

- * 在 Vim 的一个垂直或水平的分割窗口中显示一个文件中定义的 tags(函数，类，结构，
变量，等)

- * 在 GUI Vim 中，可以选择把 tags 显示在下拉菜单和弹出菜单中
- * 当你在多个源文件/缓冲区间切换时，taglist 窗口会自动进行相应地更新。
当你打开新文件时，新文件中定义的 tags 会被添加到已经存在的文件列表中，并且所有文件中定义的 tags 会以文件名来分组显示
- * 当你在 taglist 窗口中选中一个 tag 名时，源文件中的光标会自动跳转到该 tag

的定

义处

- * 自动高亮当前的 tag 名
- * 按类型分组各 tag，并且将各组显示在一个可折叠的树形结构中
- * 可以显示 tag 的原始类型和作用域
- * 在 taglist 窗口可选择显示 tag 的原始类型替代 tag 名
- * tag 列表可以按照 tag 名，或者时间进行排序
- * 支持以下语言的源文件：Assembly, ASP, Awk, Beta, C, C++, C#, Cobol, Eiffel, Erlang, Fortran, HTML, Java, Javascript, Lisp, Lua, Make, Pascal, Perl, PHP, Python, Rexx, Ruby, Scheme, Shell, Slang, SML, Sql, TCL, Verilog, Vim and Yacc.
- * 可以很简单的扩展支持新的语言。对新语言支持的修改也很简单。
- * 提供了一些函数，可以用来在 Vim 的状态栏或者在窗口的标题栏显示当前的 tag

名

- * taglist 中的文件和 tags 的列表可以在被保存和在 vim 会话间加载
- * 提供了一些用来取得 tag 名和原始类型的命令
- * 在控制台 vim 和 GUI vim 中都可以使用
- * 可以和 winmanager 插件一起使用。winmanager 插件可以让你同时使用文件浏览

器，

缓冲区浏览器和 taglist 插件，就像一个 IDE 一样。

- * 可以在 Unix 和 MS-Windows 系统中使用

首先请先在你的 ~/.vimrc 文件中添加下面两句：

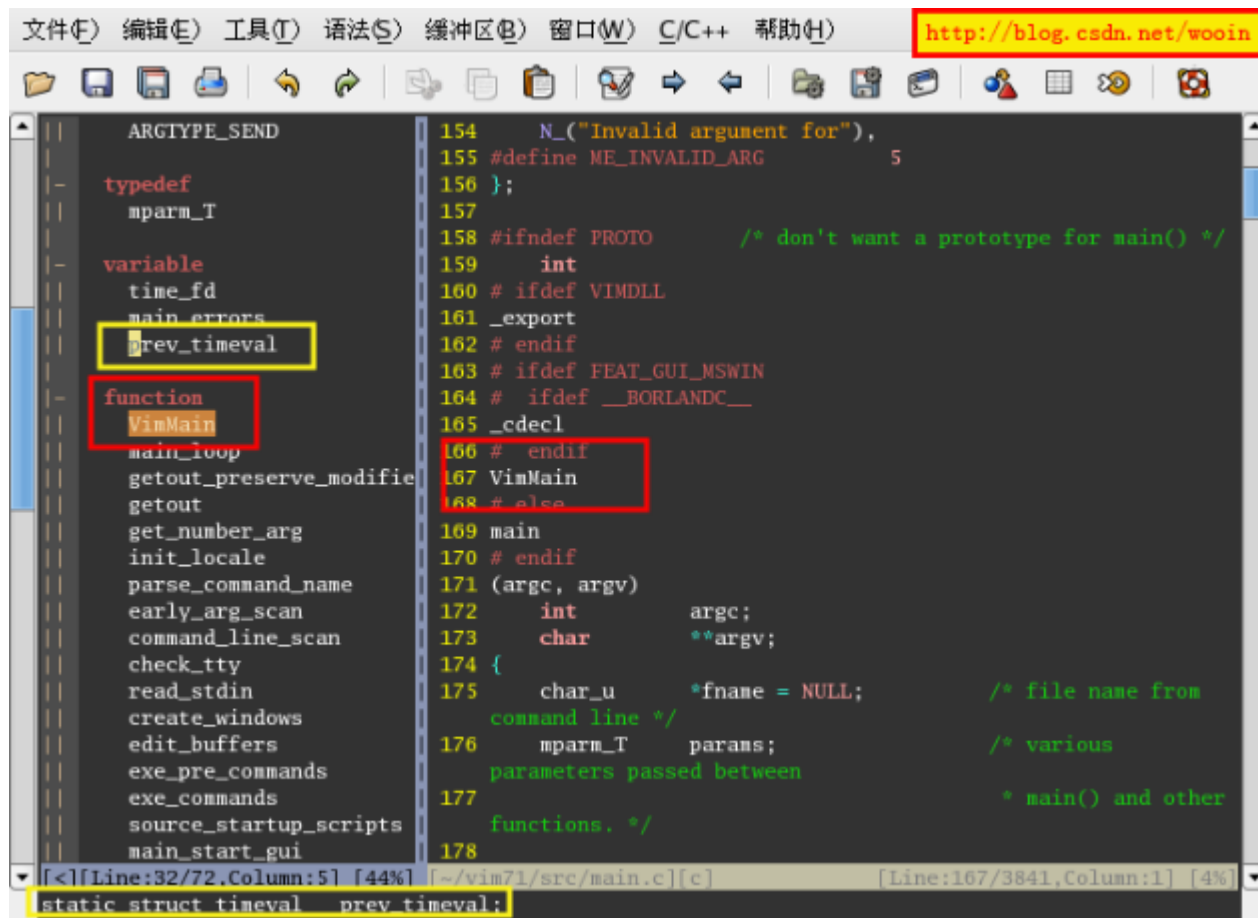
```
let Tlist_Show_One_File=1
let Tlist_Exit_OnlyWindow=1
```

此时用 vim 打开一个 c 源文件试试：

```
$ vim ~/vim/src/main.c
```


进入 vim 后用下面的命令打开 taglist 窗口，如图 5：

```
:Tlist
```



(--- 图 5 ---)

左边的窗口就是前面介绍的 TagList 窗口，其中列出了 main.c 文件中的 tag，并且按照 “typedef”，“variable”，“function” 等进行了分类。将光标移到 VimMain 上，如图中左边红色的方框，按下回车后，源程序会自动跳转到 VimMain 的定义处，如图中右边的红色方框。这就是 TagList 最基本也是最常用的操作。再教你一个常用的操作，你在浏览 TagList 窗口时，如果还不想让源码跳转，但是想看看 tag 在源码中完整的表达，可以将光标移到你

想要看的 tag 上，如图中上边黄色的方框，然后按下空格键，在下面的命令栏中，如图下边黄色的方框，会显示该 tag 在源码中完整的写法，而不会跳转到源码处。

TagList 插件我就介绍到这里，其实它还有很多用法和设置，我没法一一地教你了，好在 TagList 有一份详细的帮助手册，用下面的命令打开手册，好好学习一下吧：

```
:help taglist.txt
```

7. 文件浏览器和窗口管理器 -- 插件: WinManager

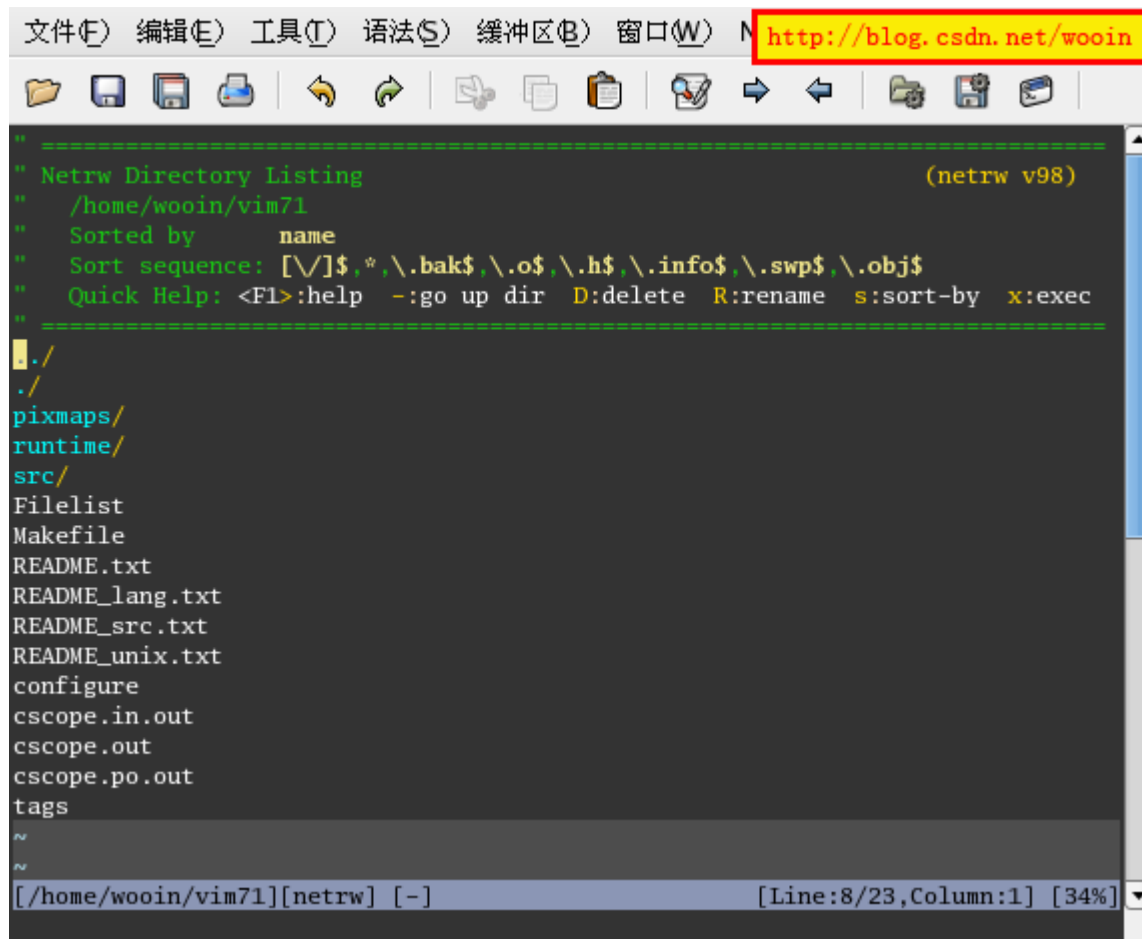
下载地址	http://www.vim.org/scripts/script.php?script_id=95
版本	2. x
安装	在 ~/.vim 目录下解压 winmanager.zip
手册	:help winmanager

在图 1 中大家可以看到在图标 2 标识的地方是一个文件浏览器，里面列出了当前目录中的文件，可以通过这个浏览器来浏览工程中的源文件，是不是越来越像常见的 IDE 了，当光标停在某个文件或文件夹的时候，按下回车，可以打开该文件或文件夹。

这个东东是怎么调出来的呢？其实这个也是由插件实现的，这个插件是 netrw.vim，只不过你不用下载和安装，这个插件已经是标准的 vim 插件，已经随 vim 一起安装进你的系统里了，现在先简单演示一下，进入“~/vim71”文件夹后运行 vim，然后在 vim 中运行命令：

```
:e ~/vim71
```

你将在 vim 看到如下图所示的界面：



(--- 图 6 ---)

在该界面上你可以用下面的一些命令来进行常用的目录和文件操作：

<F1>	显示帮助
<cr>	如果光标下是目录，则进入该目录；如果光标下文件，则打开该文件
-	返回上级目录
c	切换 vim 当前工作目录正在浏览的目录
d	创建目录

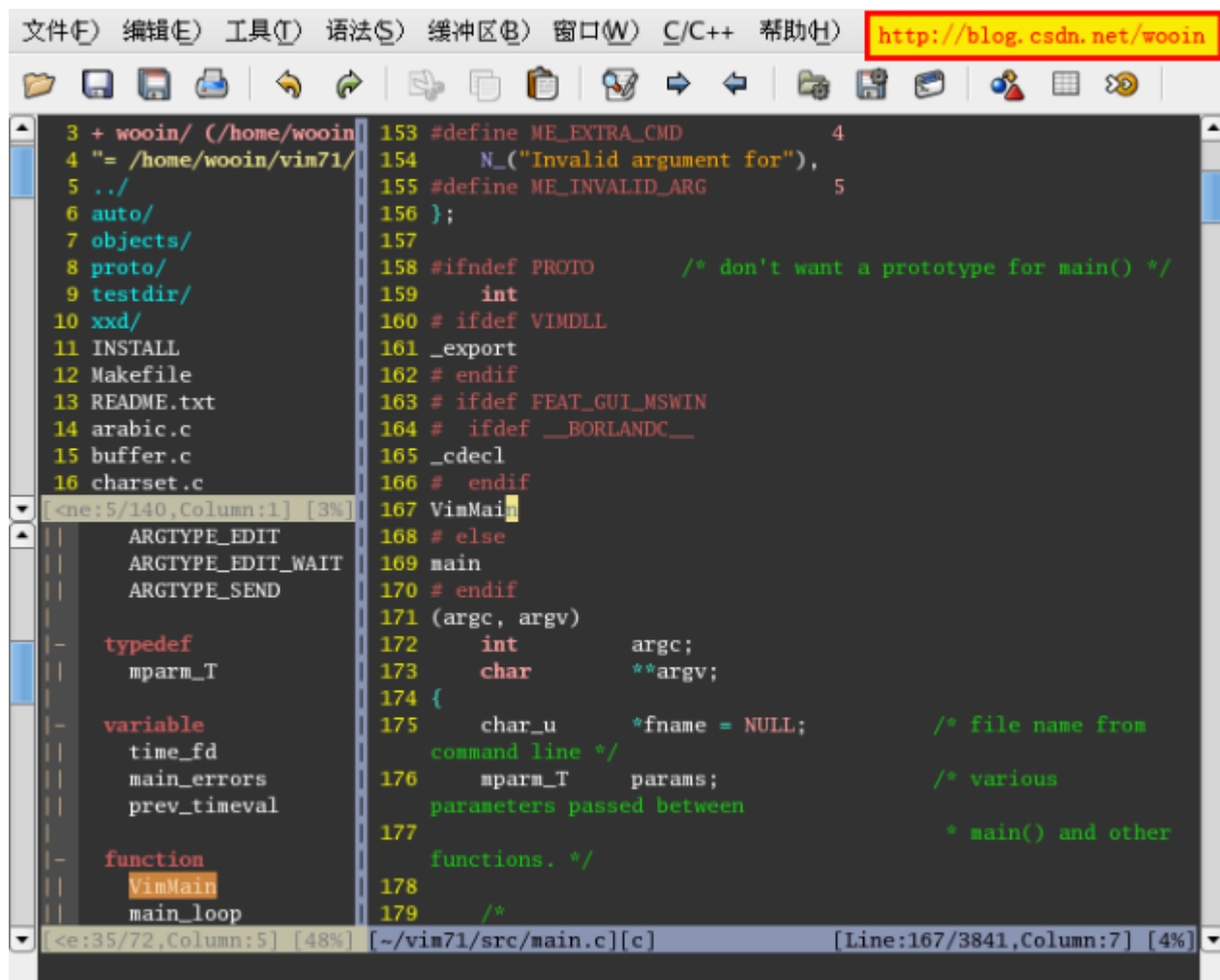
D	删除目录或文件
i	切换显示方式
R	文件或目录重命名
s	选择排序方式
x	定制浏览方式，使用你指定的程序打开该文件

我这里不是教你怎么用 netrw.vim 插件，而是要教你通过 WinManager 插件来将 TagList 窗口和 netrw 窗口整合起来，就像图 1 中的图标 2 和 3 组成的那个效果

现在在你的 ~/.vimrc 中增加下面两句

```
let g:winManagerWindowLayout='FileExplorer|TagList'  
nmap wm :WMToggle<cr>
```

然后重启 vim，打开 ~/vim71/src/main.c，在 normal 状态下输入“wm”，你将看到图 7 的样子：



(--- 图 7 ---)

其中左上边是netrw窗口，左下边是TagList窗口，当再次输入“wm”命令时这两个窗口又关闭了。

WinManager 的功能主要就是我介绍的这些，但是它还有其他一些高级功能，还可以支持其他几个插件，如果你觉得我介绍的还不够你用，建议你把它的手册好好研究一下，用下面的命令可以调出帮助手册：

```
:help winmanager
```

8. Cscope 的使用

这下更厉害了，用 Cscope 自己的话说 - “你可以把它当做是超过频的 ctags”，其功能和强大程度可见一斑吧，关于它的介绍我就不详细说了，如果你安装好了前文介绍的中文帮助手册，用下面的命令看看介绍吧：

```
:help if_cscop.txt
```

我在这里简单摘抄一点，供还在犹豫的朋友看看：

Cscope 是一个交互式的屏幕下使用的工具，用来帮助你：

- * 无须在厚厚的程序清单中翻来翻去就可以认识一个 C 程序的工作原理。
- * 无须熟悉整个程序就可以知道清楚程序 bug 所要修改的代码位置。
- * 检查提议的改动（如添加一个枚举值）可能会产生的效果。
- * 验证所有的源文件都已经作了需要的修改；例如给某一个现存的函数添加一个参数。
- * 在所有相关的源文件中对一个全局变量改名。
- * 在所有相关的位置将一个常数改为一个预处理符号。

它被设计用来回答以下的问题：

- * 什么地方用到了这个符号？
- * 这是在什么地方定义的？
- * 这个变量在哪里被赋值？
- * 这个全局符号的定义在哪里？
- * 这个函数在源文件中的哪个地方？
- * 哪些函数调用了这个函数？
- * 这个函数调用了哪些函数？
- * 信息 “out of space” 从哪来？
- * 这个源文件在整个目录结构中处于什么位置？
- * 哪些文件包含这个头文件？

安装 Cscope：

如果你的系统中有 cscope 命令，则可以跳过这一小段，如果没有，就先跟着我一起安装一个吧。

在 Cscope 的主页：<http://cscope.sourceforge.net> 下载一个源码包，解压后编译安装：

```
# ./configure
# make
# make install          // 需要 root
                          权限
```

先在 ~/.vimrc 中增加一句：

```
:set cscopequickfix=s-,c-,d-,i-,t-,e-
```

这个是设定是否使用 quickfix 窗口来显示 cscope 结果，用法在后面会说到。

跟 Ctags 一样，要使用其功能必须先为你的代码生成一个 cscope 的数据库，在项目的根目录运行下面的命令：

```
$ cd /home/wooin/vim71/
$ cscope -Rbq
# 此后会生成三个文件
$ ll cscope.*
-rw-rw-r-- 1 wooin wooin 1.1M 2007-09-30 10:56 cscope.in.out
-rw-rw-r-- 1 wooin wooin 6.7M 2007-09-30 10:56 cscope.out
-rw-rw-r-- 1 wooin wooin 5.1M 2007-09-30 10:56 cscope.po.out
# 打开文件，开始 Coding
$ cd src
$ vi main.c
```

进入 vim 后第一件事是要把刚才生成的 cscope 文件导入到 vim 中来，用下面的命令：

```
:cs add /home/wooin/vim71/cscope.out /home/wooin/vim71
```

上面这条命令很重要，必须写全，不能只写前半句：

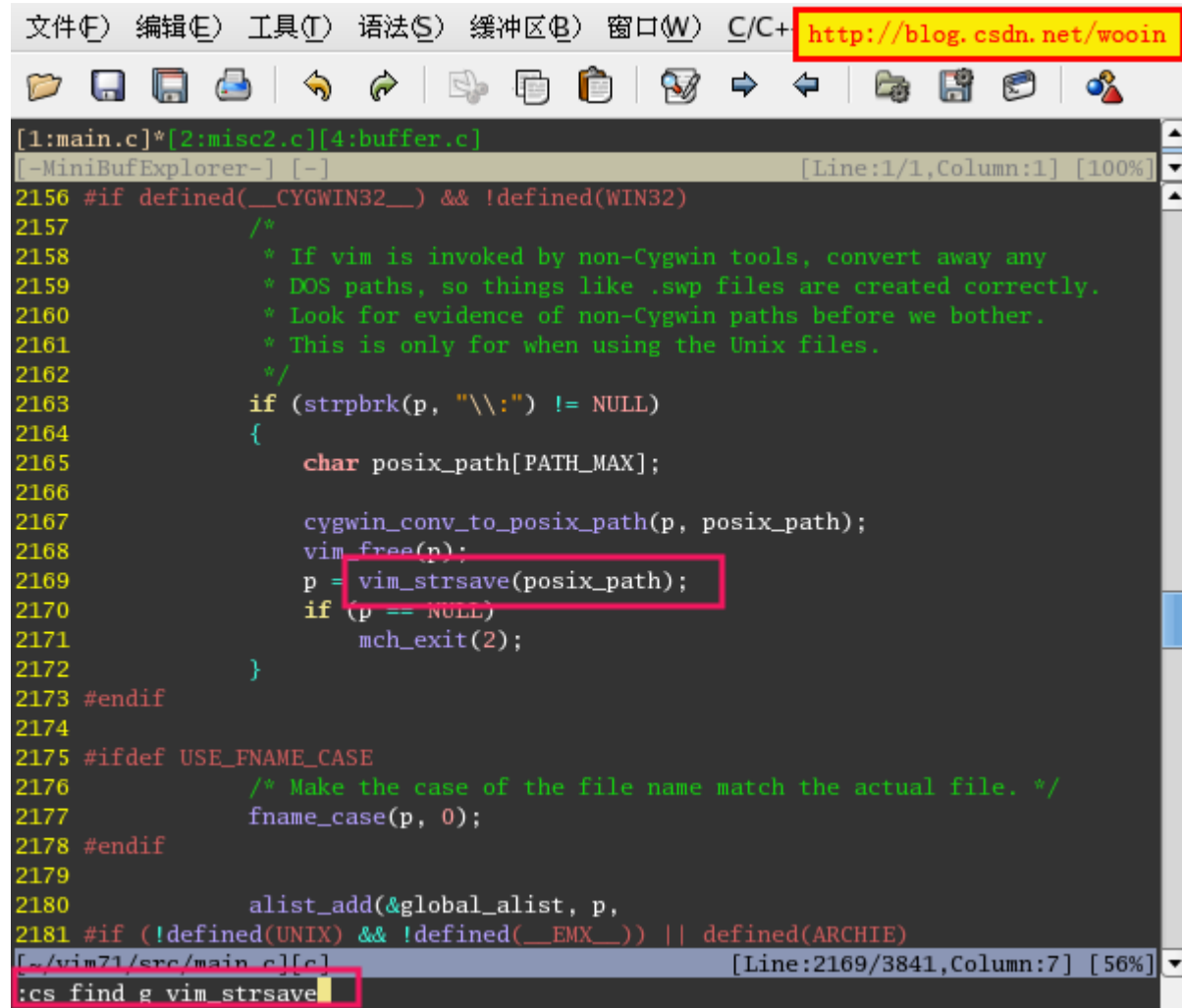
```
:cs add /home/wooin/vim71/cscope.out
```

因为源码是多级目录的，如果这样写，cscope 是无法在子目录中的源码中工作的，当然，如果你的源码都在同一级目录中就无所谓了。如果你要经常用 cscope 的话，可以把上面那句加到 ~/.vimrc 中去。

下面我们来操练一下，查找函数 vim_strsave() 的定义，用命令：

```
:cs find g vim_strsave
```

如下图:



```
[1:main.c]*[2:misc2.c][4:buffer.c]
[-MiniBufExplorer-] [-] [Line:1/1,Column:1] [100%]
2156 #if defined(__CYGWIN32__) && !defined(WIN32)
2157 /*
2158  * If vim is invoked by non-Cygwin tools, convert away any
2159  * DOS paths, so things like .swp files are created correctly.
2160  * Look for evidence of non-Cygwin paths before we bother.
2161  * This is only for when using the Unix files.
2162  */
2163 if (strpbrk(p, "\\:") != NULL)
2164 {
2165     char posix_path[PATH_MAX];
2166
2167     cygwin_conv_to_posix_path(p, posix_path);
2168     vim_free(p);
2169     p = vim_strsave(posix_path);
2170     if (p == NULL)
2171         mch_exit(2);
2172 }
2173 #endif
2174
2175 #ifdef USE_FNAME_CASE
2176 /* Make the case of the file name match the actual file. */
2177 fname_case(p, 0);
2178 #endif
2179
2180 alist_add(&global_alist, p,
2181 #if (!defined(UNIX) && !defined(__EMX__)) || defined(ARCHIE)
[-/vim71/src/main.c][c] [Line:2169/3841,Column:7] [56%]
:cs find g vim_strsave
```

(--- 图 8 ---)

按下回车后会自动跳转到 `vim_strsave()` 的定义处。此时你肯定会说 Ctags 也可以做到这个呀，那么下面说的这个 Ctags 就无法做到了，我想查找 `vim_strsave()` 到底在那些地方被调用过了，用命令：

```
:cs find c vim_strsave
```

按下回车后 vim 会自动跳转到第一个符合要求的地方，并且在命令栏显示有多少符合要求的

文件(F) 编辑(E) 工具(T) 语法(S) 缓冲区(B) 窗口(W) C/C+ <http://blog.csdn.net/wooin>

```
[1:main.c][2:misc2.c][4:buffer.c]*
[-MiniBufExplorer-] [-] [Line:1/1,Column:22] [100%]
1572 {
1573     buf = (buf_T *)alloc_clear((unsigned)sizeof(buf_T));
1574     if (buf == NULL)
1575     {
1576         vim_free(ffname);
1577         return NULL;
1578     }
1579 }
1580
1581 if (ffname != NULL)
1582 {
1583     buf->b_ffname = ffname;
1584     buf->b_sfname = vim_strsave(sfname);
1585 }
1586
1587 clear_wininfo(buf);
1588 buf->b_wininfo = (wininfo_T *)alloc_clear((unsigned)sizeof(wininfo_T));
1589
1590 if ((ffname != NULL && (buf->b_ffname == NULL || buf->b_sfname == NULL))
1591     || buf->b_wininfo == NULL)
1592 {
1593     vim_free(buf->b_ffname);
1594     buf->b_ffname = NULL;
1595     vim_free(buf->b_sfname);
1596     buf->b_sfname = NULL;
1597     if (buf != curbuf)
1598         buf->b_wininfo = NULL;
1599 }
```

如果自动跳转的位置你不满意，想看其他的结果，可以用下面的命令打开 QuickFix 窗口：

如图:


```
文件(F) 编辑(E) 工具(T) 语法(S) 缓冲区(B) 窗口(W) C/C+ http://blog.csdn.net/wooin
[1:main.c][2:misc2.c][4:buffer.c]*
[-MiniBufExplorer-] [-] [Line:1/1,Column:22] [100%]
1577         return NULL;
1578     }
1579 }
1580
1581 if (ffname != NULL)
1582 {
1583     buf->b_ffname = ffname;
1584     buf->b_sfname = vim_strsave(sfname);
1585 }
1586
1587 clear_winfo(buf);
1588 buf->b_winfo = (winfo_T *)alloc_clear((unsigned)sizeof(winfo_T));
1589
1590 if ((ffname != NULL && (buf->b_ffname == NULL || buf->b_sfname == NULL))
1591     || buf->b_winfo == NULL)
1592
~/vim71/src/buffer.c[c] [Line:1584/5517,Column:2] [28%]
1 buffer.c|1584| <<buflist_new>> buf->b_sfname = vim_strsave(sfname);
2 buffer.c|2190| <<ExpandBufnames>> p = vim_strsave(p);
3 buffer.c|2610| <<setfname>> sfname = vim_strsave(sfname);
4 buffer.c|2663| <<buf_set_name>> buf->b_ffname = vim_strsave(name);
5 buffer.c|3240| <<ti_change>> *last = vim_strsave(str);
6 buffer.c|4182| <<fix_fname>> fname = vim_strsave(fname);
7 buffer.c|4834| <<ex_buffer_all>> s = linecopy = vim_strsave(s);
8 charset.c|1913| <<backslash_halve_save>> res = vim_strsave(p);
9 diff.c|537| <<diff_check_unchanged>> line_org = vim_strsave(ml_get_buf(tp-
>tp_diffbuf[i_org],
[[Quickfix List]][qf] [-] [Line:1/378,Column:1] [0%]
:CW
```

(--- 图 10 ---)

这时你就可以慢慢挑选了`_^`

cscope 的主要功能是通过同的子命令“find”来实现的

“cscope find”的用法:

cs find c|d|e|f|g|i|s|t name

0 或 s	查找本 C 符号(可以跳过注释)
-------	------------------

1 或 g	查找本定义
2 或 d	查找本函数调用的函数
3 或 c	查找调用本函数的函数
4 或 t	查找本字符串
6 或 e	查找本 egrep 模式
7 或 f	查找本文件
8 或 i	查找包含本文件的文件

如果每次查找都要输入一长串命令的话还真是件讨人厌的事情，Cscope 的帮助手册中推荐了一些快捷键的用法，下面是其中一组，也是我用的，将下面的内容添加到 ~/.vimrc 中，并重启 vim：

```
nmap <C->s :cs find s <C-R>=expand("<cword>")<CR><CR>
nmap <C->g :cs find g <C-R>=expand("<cword>")<CR><CR>
nmap <C->c :cs find c <C-R>=expand("<cword>")<CR><CR>
nmap <C->t :cs find t <C-R>=expand("<cword>")<CR><CR>
nmap <C->e :cs find e <C-R>=expand("<cword>")<CR><CR>
nmap <C->f :cs find f <C-R>=expand("<cfile>")<CR><CR>
nmap <C->i :cs find i ^<C-R>=expand("<cfile>")<CR>$<CR>
nmap <C->d :cs find d <C-R>=expand("<cword>")<CR><CR>
```

当光标停在某个你要查找的词上时，按下<C->g，就是查找该对象的定义，其他的同理。按这种组合键有一点技巧，按了<C->后要马上按下下一个键，否则屏幕一闪就回到 nomal 状态了

<C->g 的按法是先按"Ctrl+Shift+-"，然后很快再按"g"

很奇怪，其中的这句：

```
nmap <C->i :cs find i ^<C-R>=expand("<cfile>")<CR>$<CR>
```

在我的 vim 中无法工作，但是我改成：

```
nmap <C->i :cs find i <C-R>=expand("<cfile>")<CR><CR>
```

就可以正常工作了，不知道是什么原因？有哪位朋友知道请告诉我。

cscope 的其他功能你可以通过帮助手册自己慢慢学习

reset : 重新初始化所有连接。

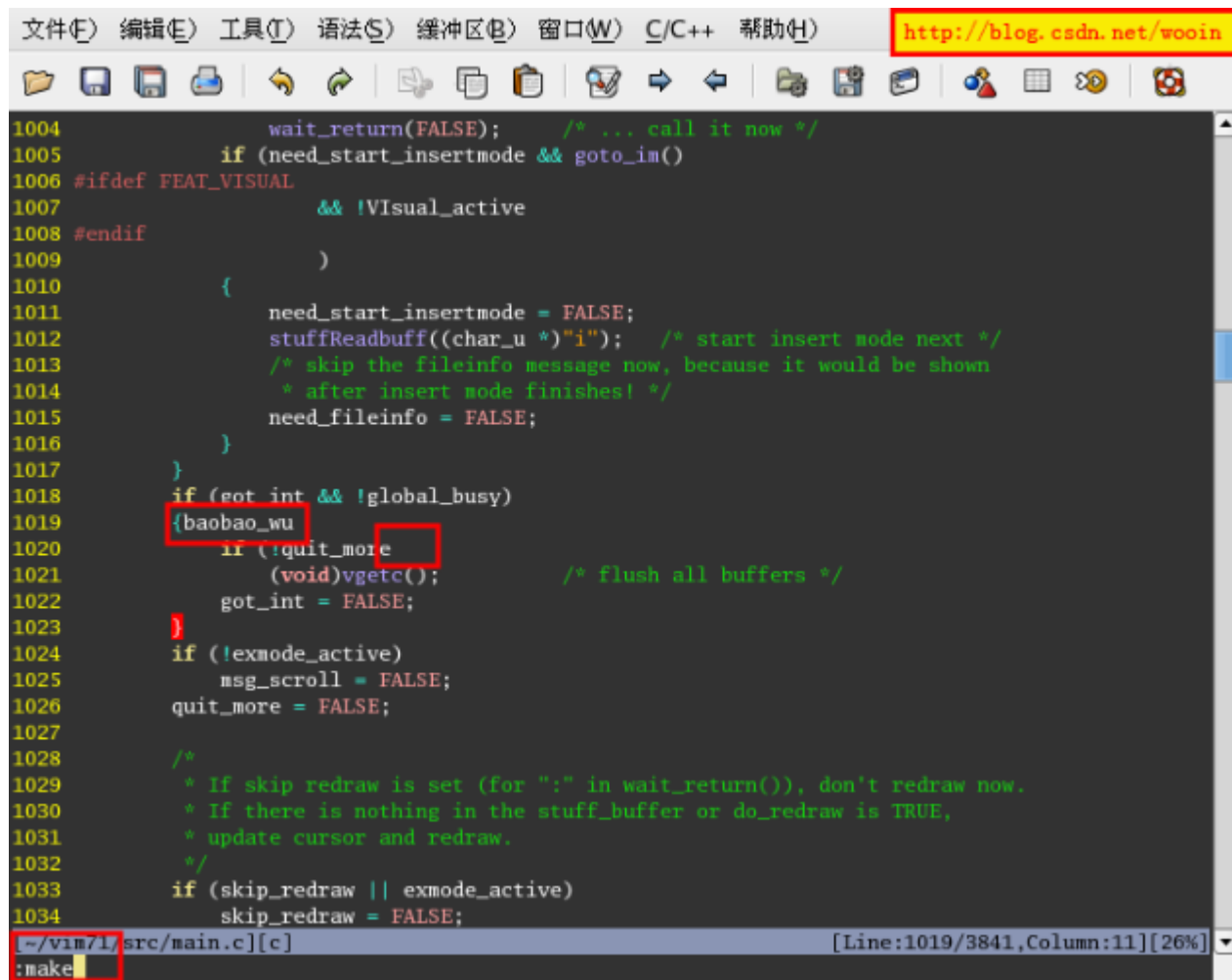
用法 : cs reset

9. QuickFix 窗口

在上一节的图 10 中大家可以看到在窗口下面有一个显示查询结果的窗口，这个窗口中列出了查询命令的查询结果，用户可以从这个窗口中选择每个结果进行查看，这个窗口叫“QuickFix”窗口，以前也是一个 vim 的插件来的，只不过现在成了 vim 的标准插件，不用你在去安装了，QuickFix 窗口的主要作用就是上面看到的那个功能：输出一些供选择的结果，可以被很多命令调用，更详细的介绍和使用方法请用下面的命令打开 QuickFix 的手册来学习吧：

```
:help quickfix
```

这里我一个常用的例子来再介绍一种 QuickFix 窗口的使用方法。这个例子是要模仿平时我们编程时，当编译出错时，QuickFix 会把出错的信息列出来，供我们一条条地查看和修改。首先还是用 vim 打开~/vim71/src/main.c，事先最好先编译过 vim71，否则一会儿编译的时候有点慢，或者你也可以自己写一个小的有错误的程序来跟着我做下面的步骤，见下图：



```
1004     wait_return(FALSE);    /* ... call it now */
1005     if (need_start_insertmode && goto_im())
1006 #ifdef FEAT_VISUAL
1007         && !Visual_active
1008 #endif
1009     )
1010     {
1011         need_start_insertmode = FALSE;
1012         stuffReadbuff((char_u *)"i"); /* start insert mode next */
1013         /* skip the fileinfo message now, because it would be shown
1014          * after insert mode finishes! */
1015         need_fileinfo = FALSE;
1016     }
1017 }
1018 if (got_int && !global_busy)
1019 {baobao_wu
1020     if (!quit_more
1021         (void)vgetc(); /* flush all buffers */
1022     got_int = FALSE;
1023 }
1024 if (!exmode_active)
1025     msg_scroll = FALSE;
1026 quit_more = FALSE;
1027
1028 /*
1029  * If skip redraw is set (for ":" in wait_return()), don't redraw now.
1030  * If there is nothing in the stuff_buffer or do_redraw is TRUE,
1031  * update cursor and redraw.
1032  */
1033 if (skip_redraw || exmode_active)
1034     skip_redraw = FALSE;
1035
1036 [Line:1019/3841,Column:11][26%]
```

~/vim/l1/src/main.c[c]
:make

(--- 图 11 ---)

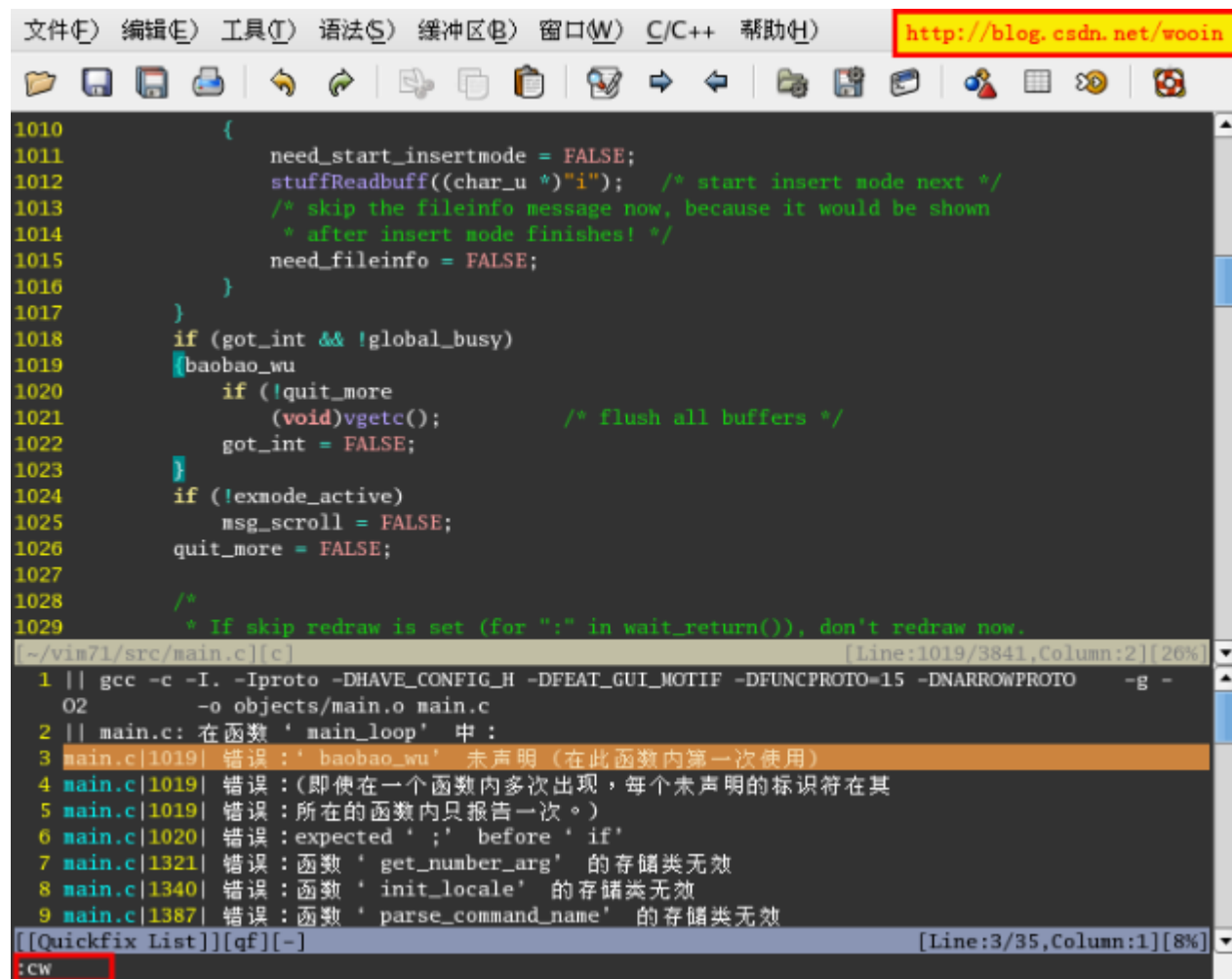
我们修改一下 main.c，人为地造成几处错误，在第 1019 行增加了一个 baobao_wu 的没有任何定义的字符串，删除了第 1020 行最后的一个括号“}”，然后用下面的命令进行编译：

```
:make
```

显然编译会报很多错误，当编译结束并退出到源码界面时，刚才编译器报的错误都已经看不到了，但是我们可以用 QuickFix 窗口再将错误信息找出来，用下面的命令调出 QuickFix 窗口：

:CW

此时你就可以看如下图所示的 QuickFix 窗口了：



(--- 图 12 ---)

在下面的 QuickFix 窗口中我们可以找到每一个编译错误，同样你可以用鼠标点击每一条记录，代码会马上自动跳转到错误处，你还可以用下面的命令来跳转：

```
:cn          // 切换到下
一个结果
```

```
:cp // 切换到上  
一个结果
```

如果你经常使用这两个命令，你还可以给他们设定快捷键，比如在`~/.vimrc`中增加：

```
nmap <F6> :cn<cr>  
nmap <F7> :cp<cr>
```

其还有其他的命令/插件也会用到 QuickFix 窗口，但是用法基本上的都是类似的，本文后面还会用到 QuickFix 窗口，接着往下看吧。

10. 快速浏览和操作 Buffer -- 插件：MiniBufExplorer

下载地址	http://www.vim.org/scripts/script.php?script_id=159
版本	6.3.2
安装	将下载的 minibufexpl.vim 文件丢到 <code>~/.vim/plugin</code> 文件夹中即可
手册	在 minibufexpl.vim 文件的头部

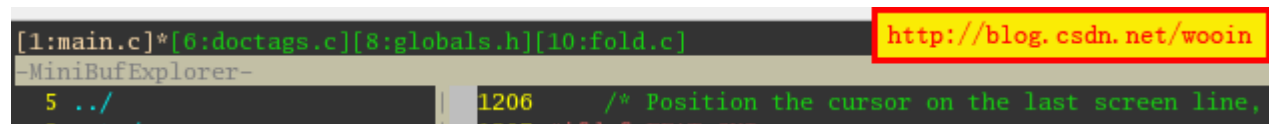
在编程的时候不可能永远只编辑一个文件，你肯定会打开很多源文件进行编辑，如果每个文件都打开一个 vim 进行编辑的话那操作起来将是多麻烦啊，所以 vim 有 buffer(缓冲区)的概念，可以看 vim 的帮助：

```
:help buffer
```

vim 自带的 buffer 管理工具只有 `:ls`，`:bnext`，`:bdelete` 等的命令，既不好用，又不直观。现在隆重向你推荐一款 vim 插件(plugin)：MiniBufExplorer

使用方法：

重新启动 vim，当你只编辑一个 buffer 的时候 MiniBufExplorer 派不上用场，当你打开第二个 buffer 的时候，MiniBufExplorer 窗口就自动弹出来了，见下图：



(--- 图 13 ---)

上面那个狭长的窗口就是 MiniBufExplorer 窗口，其中列出了当前所有已经打开的 buffer，当你把光标置于这个窗口时，有下面几个快捷键可以用：

<Tab>	向前循环切换到每个 buffer 名上
<S-Tab>	向后循环切换到每个 buffer 名上
<Enter>	在打开光标所在的 buffer
d	删除光标所在的 buffer

以下的两个功能需要在 ~/.vimrc 中增加:

```
let g:miniBufExplMapCTabSwitchBufs = 1
```

<C-Tab>	向前循环切换到每个 buffer 上, 并在但前窗口打开
<C-S-Tab>	向后循环切换到每个 buffer 上, 并在但前窗口打开

如果在 ~/.vimrc 中设置了下面这句:

```
let g:miniBufExplMapWindowNavVim = 1
```

则可以用<C-h, j, k, l>切换到上下左右的窗口中去, 就像:

C-w, h j k l 向“左, 下, 上, 右”切换窗口.

在 ~/.vimrc 中设置:

```
let g:miniBufExplMapWindowNavArrows = 1
```

是用<C-箭头键>切换到上下左右窗口中去

11. c/h 文件间相互切换 — 插件: A

下载地址	http://www.vim.org/scripts/script.php?script_id=31
版本	
安装	将 a.vim 放到 ~/.vim/plugin 文件夹中
手册	无

下面介绍它的用法:

作为一个 C 程序员, 日常 Coding 时在源文件与头文件间进行切换是再平常不过的事了, 直接用 vim 打开其源/头文件其实也不是什么麻烦事, 但是只用一个按键就切换过来了, 这是

多么贴心的功能啊....

安装好 a.vim 后有下面的几个命令可以用了：

:A	在新 Buffer 中切换到 c/h 文件
:AS	横向分割窗口并打开 c/h 文件
:AV	纵向分割窗口并打开 c/h 文件
:AT	新建一个标签页并打开 c/h 文件

其他还有一些命令，你可以在它的网页上看看，我都没用过，其实也都是大同小异，找到自己最顺手的就行了。

我在 ~/.vimrc 中增加了一句：

```
nnoremap <silent> <F12> :A<CR>
```

意思是按 F12 时在一个新的 buffer 中打开 c/h 文件，这样在写程序的时候就可以不假思索地在 c/h 文件间进行切换，减少了按键的次数，思路也就更流畅了，阿弥陀佛....

12. 在工程中查找 -- 插件：Grep

下载地址	http://www.vim.org/scripts/script.php?script_id=311
版本	1.8
安装	把 grep.vim 文件丢到 ~/.vim/plugin 文件夹就好了
手册	在 grep.vim 文件头部

下面介绍它的用法：

vim 有自己的查找功能，但是跟 shell 中的 grep 比起来还是有些差距的，有时 Coding 正火急火燎的时候，真想按下 F3，对光标所在的词来个全工程范围的 grep，不用敲那些繁琐的命令，现在福音来了，跟我同样懒的人不在少数，在 grep.vim 脚本的前部可以找到一些说明文档：

:Grep	按照指定的规则在指定的文件中查找
:Rgrep	同上，但是是递归的 grep
:GrepBuffer	在所有打开的缓冲区中查找

:Bgrep	同上
:GrepArgs	在 vim 的 argument filenames (:args) 中查找
:Fgrep	运行 fgrep
:Rfgrep	运行递归的 fgrep
:Egrep	运行 egrep
:Regrep	运行递归的 egrep
:Agrep	运行 agrep
:Ragrep	运行递归的 agrep

上面的命令是类似这样调用的:

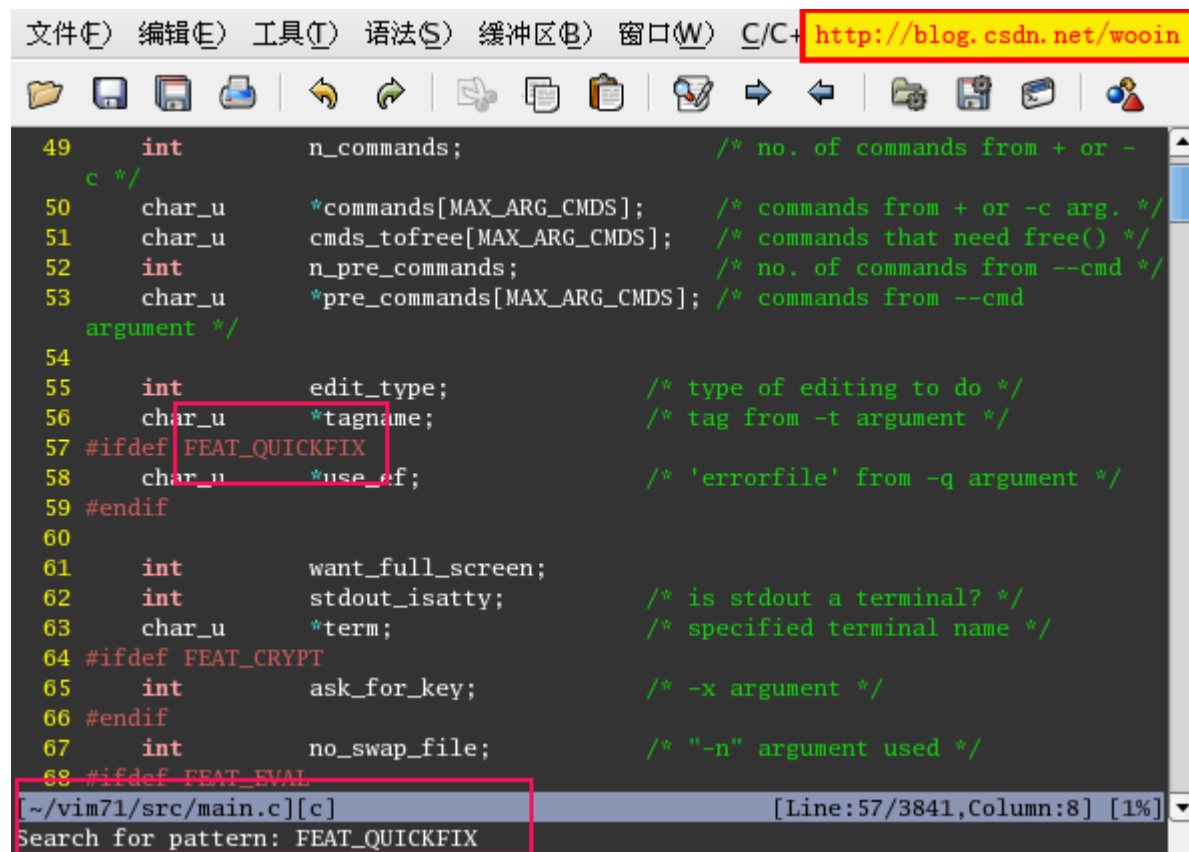
:Grep	[<grep_options>] [<search_pattern> [<file_name(s)>]]
:Rgrep	[<grep_options>] [<search_pattern> [<file_name(s)>]]
:Fgrep	[<grep_options>] [<search_pattern> [<file_name(s)>]]
:Rfgrep	[<grep_options>] [<search_pattern> [<file_name(s)>]]
:Egrep	[<grep_options>] [<search_pattern> [<file_name(s)>]]
:Regrep	[<grep_options>] [<search_pattern> [<file_name(s)>]]
:Agrep	[<grep_options>] [<search_pattern> [<file_name(s)>]]
:Ragrep	[<grep_options>] [<search_pattern> [<file_name(s)>]]
:GrepBuffer	[<grep_options>] [<search_pattern>]
:Bgrep	[<grep_options>] [<search_pattern>]
:GrepArgs	[<grep_options>] [<search_pattern>]

但是我从来都不用敲上面那些命令的`^_`，因为我在`~/.vimrc`中增加了下面这句：

```
noremap <silent> <F3> :Grep<CR>
```

比如你想在`/home/wooin/vim71/src/main.c`中查找“FEAT_QUICKFIX”，则将光标移到

“FEAT_QUICKFIX”上，然后按下 F3 键，如下图：



The screenshot shows a Vim editor window with a menu bar at the top containing '文件(F)', '编辑(E)', '工具(T)', '语法(S)', '缓冲区(B)', '窗口(W)', and 'C/C+'. A red box highlights the address bar with the URL 'http://blog.csdn.net/wooin'. Below the menu bar is a toolbar with various icons. The main editor area displays C code with line numbers 49 through 68. The code includes several variable declarations and conditional compilation blocks. A red box highlights the line '57 #ifdef FEAT_QUICKFIX'. At the bottom of the editor, a status bar shows '[~/vim71/src/main.c][c]' and '[Line:57/3841,Column:8] [1%]'. Below the status bar, a search bar contains the text 'Search for pattern: FEAT_QUICKFIX'.

```
49  int      n_commands;          /* no. of commands from + or -  
c */  
50  char_u    *commands[MAX_ARG_CMDS]; /* commands from + or -c arg. */  
51  char_u    cmds_tofree[MAX_ARG_CMDS]; /* commands that need free() */  
52  int      n_pre_commands;      /* no. of commands from --cmd */  
53  char_u    *pre_commands[MAX_ARG_CMDS]; /* commands from --cmd  
argument */  
54  
55  int      edit_type;          /* type of editing to do */  
56  char_u    *tagname;         /* tag from -t argument */  
57 #ifdef FEAT_QUICKFIX  
58  char_u    *use_cf;          /* 'errorfile' from -q argument */  
59 #endif  
60  
61  int      want_full_screen;  
62  int      stdout_isatty;      /* is stdout a terminal? */  
63  char_u    *term;            /* specified terminal name */  
64 #ifdef FEAT_CRYPT  
65  int      ask_for_key;        /* -x argument */  
66 #endif  
67  int      no_swap_file;      /* "-n" argument used */  
68 #ifdef FEAT_EVAL  
[~/vim71/src/main.c][c] [Line:57/3841,Column:8] [1%]  
Search for pattern: FEAT_QUICKFIX
```

(--- 图 14 ---)

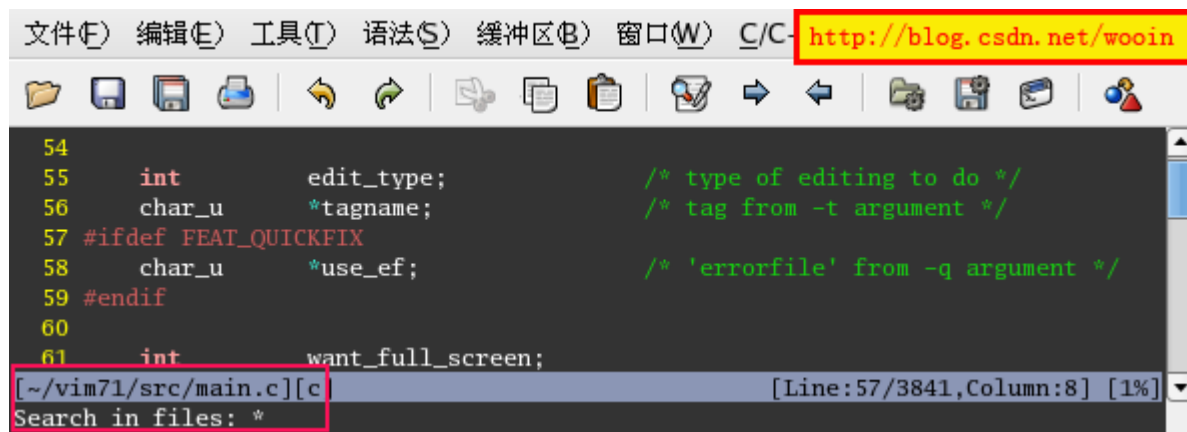
在最下面的命令行会显示：

```
Search for pattern: FEAT_QUICKFIX
```

此时你还可以编辑该行，grep 支持正则表达式，你想全词匹配的话可以改成：

```
Search for pattern: /<FEAT_QUICKFIX/>
```

然后按下回车：



```
文件(F) 编辑(E) 工具(T) 语法(S) 缓冲区(B) 窗口(W) C/C- http://blog.csdn.net/wooin
54
55     int      edit_type;          /* type of editing to do */
56     char_u    *tagname;          /* tag from -t argument */
57 #ifdef FEAT_QUICKFIX
58     char_u    *use_ef;           /* 'errorfile' from -q argument */
59 #endif
60
61     int      want_full_screen;
[~/vim71/src/main.c][c [Line:57/3841,Column:8] [1%]
Search in files: *
```

(--- 图 15 ---)

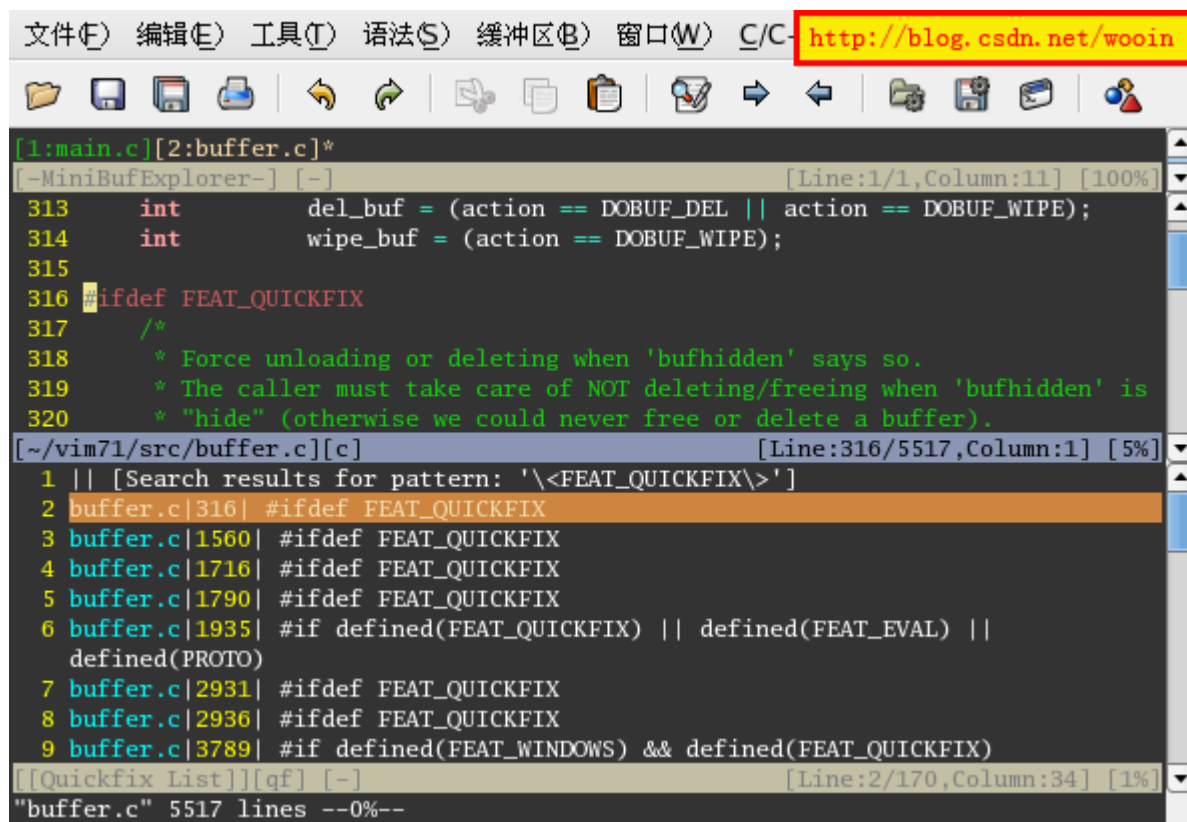
在最下面的命令行会显示:

```
Search in files: *
```

是问你搜索范围, 默认是该目录下的所有文件, 此时你还可以编辑该行, 比如你只想搜索源码文件:

```
Search in files: *.c *.h
```

然后在按下回车, 会在弹出的 QuickFix 窗口中列出所有符合条件的搜索结果, 你可以在其中查找你想要的结果, 如下图:



```
文件(F) 编辑(E) 工具(T) 语法(S) 缓冲区(B) 窗口(W) C/C- http://blog.csdn.net/wooin
[1:main.c][2:buffer.c]*
[-MiniBufExplorer-] [-] [Line:1/1,Column:11] [100%]
313     int      del_buf = (action == DOBUF_DEL || action == DOBUF_WIPE);
314     int      wipe_buf = (action == DOBUF_WIPE);
315
316 #ifdef FEAT_QUICKFIX
317 /*
318  * Force unloading or deleting when 'bufhidden' says so.
319  * The caller must take care of NOT deleting/freeing when 'bufhidden' is
320  * "hide" (otherwise we could never free or delete a buffer).
[~/vim71/src/buffer.c][c] [Line:316/5517,Column:1] [5%]
1 || [Search results for pattern: '<FEAT_QUICKFIX>']
2 buffer.c|316| #ifdef FEAT_QUICKFIX
3 buffer.c|1560| #ifdef FEAT_QUICKFIX
4 buffer.c|1716| #ifdef FEAT_QUICKFIX
5 buffer.c|1790| #ifdef FEAT_QUICKFIX
6 buffer.c|1935| #if defined(FEAT_QUICKFIX) || defined(FEAT_EVAL) ||
  defined(PROTO)
7 buffer.c|2931| #ifdef FEAT_QUICKFIX
8 buffer.c|2936| #ifdef FEAT_QUICKFIX
9 buffer.c|3789| #if defined(FEAT_WINDOWS) && defined(FEAT_QUICKFIX)
[[Quickfix List]][qf] [-] [Line:2/170,Column:34] [1%]
"buffer.c" 5517 lines --0%--
```

(--- 图 16 ---)

其实还有一些其他功能和设置，但是我都没有用过，这些功能再加上正则表达式，已经够我用了，其他的你可以在网页上看看它的文档，如果有什么惊人发现记得跟我互通有无，共同进步哦....

13. 高亮的书签 -- 插件: VisualMark

下载地址	http://www.vim.org/scripts/script.php?script_id=1026
版本	
安装	把 visualmark.vim 文件丢到 ~/.vim/plugin 文件夹就好了
手册	无

下面介绍它的用法:

vim 也和其他编辑器一样有“书签”概念，在 vim 中叫“Mark”，可以用下面的命令查看相关说明:

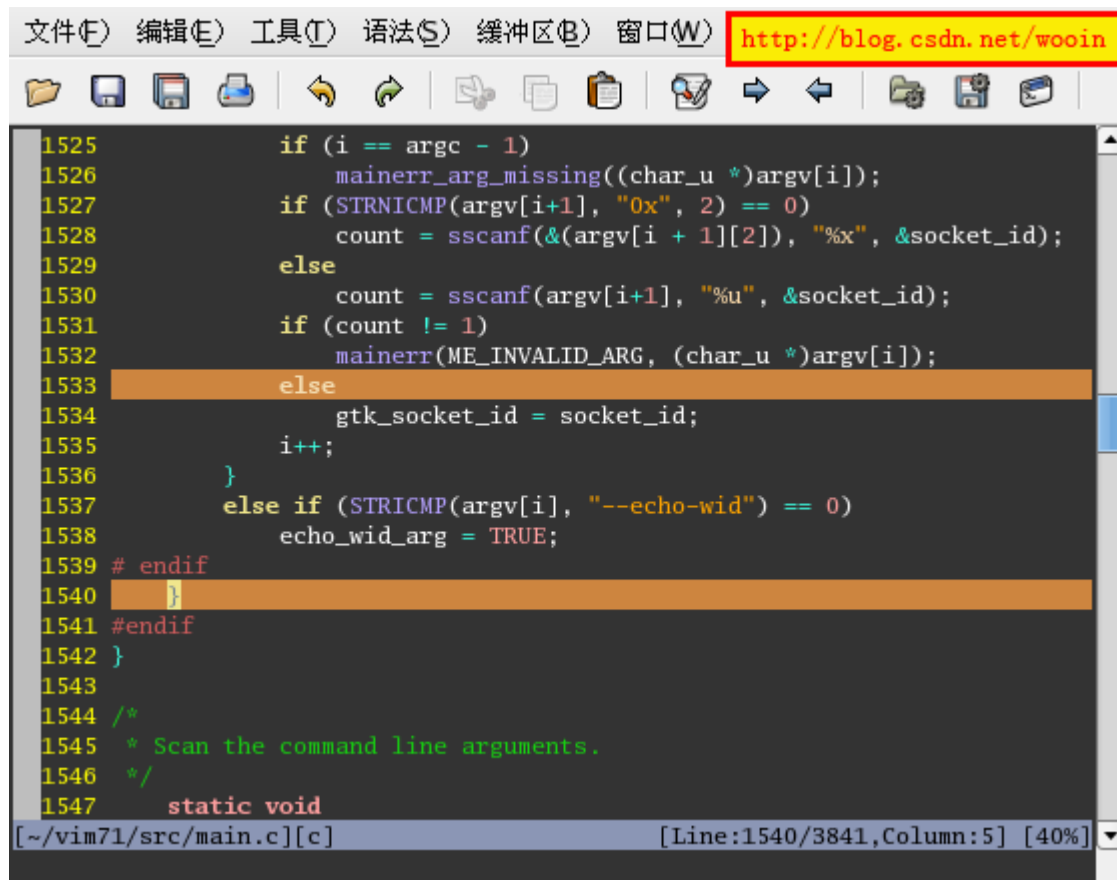
```
:help Mark
```

该“书签”有个很很大的缺点：不可见.

我下面要介绍的 Visual Mark 插件跟 vim 中的“Mark”没有什么关系，并不是使其可见，而是自己本身就是“可见的书签”，接着往下看就明白了，用作者的话说就是“类似 UltraEdit 中的书签”.

另外，网上也有使 vim 中的 Mark 可见的插件，但是我试了一下，好像没 Visual Mark 好用，我就不介绍了.

按照上面的方法安装好 Visual Mark 后，你什么也不用设置，如果是 gvim，直接在代码上按下 Ctrl+F2，如果是 vim，用“mm”，怎么样，发现光标所在的行变高亮了吧，见下图:



```
文件(F) 编辑(E) 工具(T) 语法(S) 缓冲区(B) 窗口(W) http://blog.csdn.net/wooin
1525     if (i == argc - 1)
1526         mainerr_arg_missing((char_u *)argv[i]);
1527     if (STRNICMP(argv[i+1], "0x", 2) == 0)
1528         count = sscanf(&(argv[i + 1][2]), "%x", &socket_id);
1529     else
1530         count = sscanf(argv[i+1], "%u", &socket_id);
1531     if (count != 1)
1532         mainerr(ME_INVALID_ARG, (char_u *)argv[i]);
1533     else
1534         gtk_socket_id = socket_id;
1535         i++;
1536     }
1537     else if (STRICMP(argv[i], "--echo-wid") == 0)
1538         echo_wid_arg = TRUE;
1539 # endif
1540 }
1541 #endif
1542 }
1543
1544 /*
1545  * Scan the command line arguments.
1546  */
1547 static void
```

[~/vim71/src/main.c][c] [Line:1540/3841,Column:5] [40%]

(--- 图 17 ---)

如果你设置了多个书签，你可以用 F2 键正向在期间切换，用 Shift+F2 反向在期间切换。好了，我 Visual Mark 介绍完了，够简单吧^_^。

如果你嫌书签的颜色不好看，你还可以自己定义，不过是修改这个插件脚本的源码，在目录 ~/.vim/plugin/ 中找到并打开 visualmark.vim，找到下面这段代码：

```
if &bg == "dark"    // 根据你的背景色风格来设置不同的书签颜色
    highlight SignColor ctermfg=white ctermbg=blue guifg=wheat guibg=peru
else                // 主要就是修改 guibg 的值来设置书签的颜色
    highlight SignColor ctermbg=white ctermfg=blue guibg=grey guifg=RoyalBlue3
```

```
endif
```

我还有几个不满意的地方:

- 1 这个书签不能自动保存, 关闭 vim 就没了.
- 2 切换书签时不能在不同文件间切换, 只能在同一个文件中切换

如果哪位朋友能解决这两个问题, 请一定要告诉寡人啊.... 还是用下面的地址:

Email	: lazy.fox.wu#gmail.com
Homepage	: http://blog.csdn.net/wooin

14. 自动补全

用过Microsoft Visual Studio的朋友一定知道代码补全功能, 输入一个对象名后再输入"." 或者"->", 则其成员名都可以列出来, 使 Coding 流畅了许多, 实现很多懒人的梦想, 现在我要告诉你, 这不再是 Microsoft Visual Studio 的专利了, vim 也可以做到! 下面由我来教你, 该功能要 tags 文件的支持, 并且是 ctags 5.6 版本, 可以看看前文介绍 tags 文件的章节.

我这里要介绍的功能叫"new-omni-completion(全能补全)", 你可以用下面的命令看看介绍:

```
:help new-omni-completion
```

你还需要在 ~/.vimrc 文件中增加下面两句:

```
filetype plugin indent on
```

打开文件类型检测, 加了这句才可以用智能补全

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

关掉智能补全时的预览窗口

请确定你的 Ctags 5.6 已经安装好, 并且生成的 tags 文件已经可以用了, 那么我们就要抄家伙开搞了.

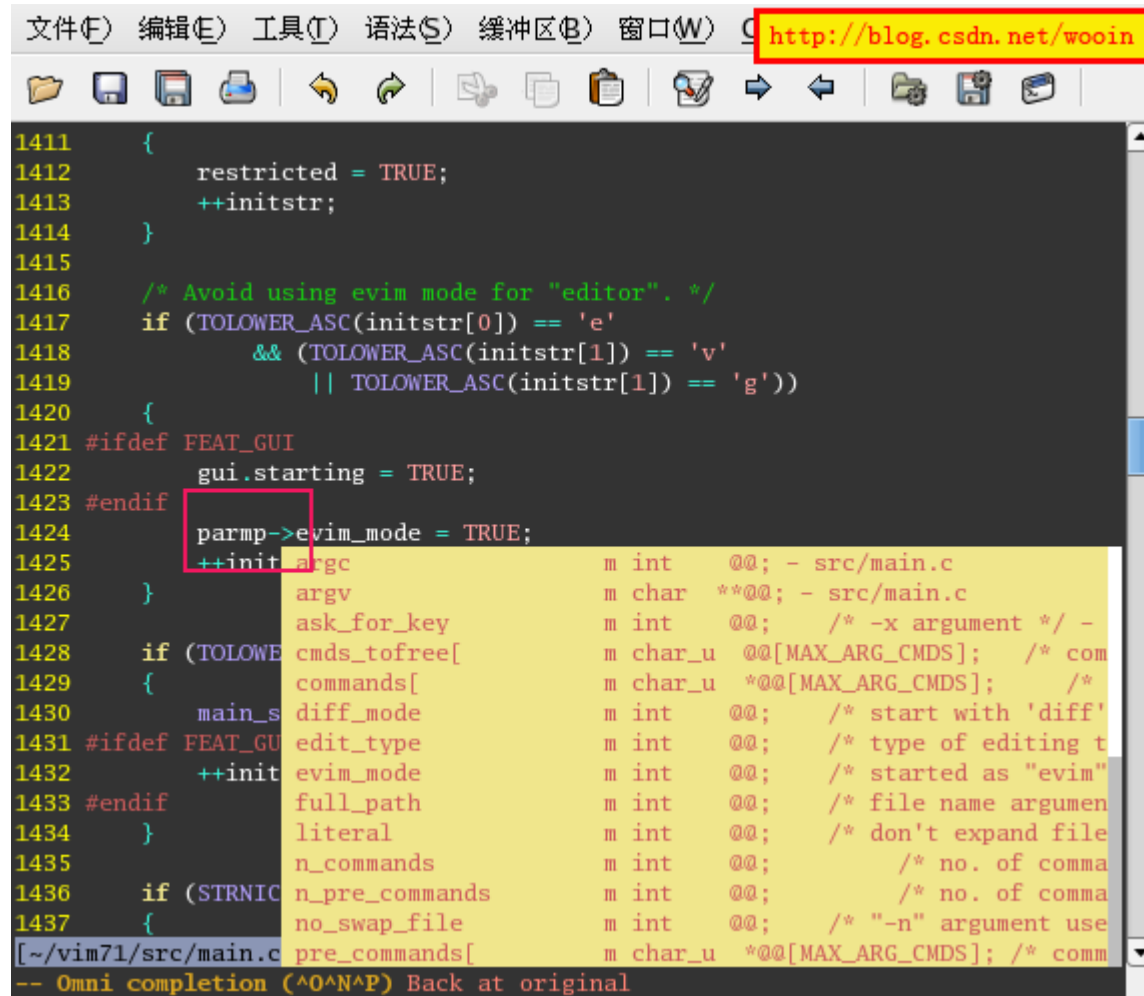
用 vim 打开源文件

```
$ vi /home/wooin/vim71/src/main.c
```

设置 tags 文件

```
:set tags=/home/wooin/vim71/tags
```

随便找一个有成员变量的对象，比如“parmp”，进入 Insert 模式，将光标放在“->”后面，然后按下“Ctrl+X Ctrl+O”，此时会弹出一个下列菜单，显示所有匹配的标签，如下图：



(--- 图 18 ---)

此时有一些快捷键可以用：

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

如果你增加了一些成员变量，全能补全还不能马上将新成员补全，需要你重新生成一下 tags 文件，但是你别重启 vim，只是重新生成一下 tags 文件就行了，这时全能补全已经可以自动补全了，还真够“全能”吧。

vim 中的其他补全方式还有：

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

15. 加速你的补全 -- 插件：SuperTab

下载地址	http://www.vim.org/scripts/script.php?script_id=1643
版本	0.43
安装	把 supertab.vim 文件丢到 ~/.vim/plugin 文件夹就好了
手册	supertab.vim 文件头部，和命令 “:SuperTabHelp”

在上面一节中你应该学会了自动补全代码的功能，按下“Ctrl+X Ctrl+O”就搞定了，如果你够懒的话肯定会说“这么麻烦啊，居然要按四个键”，不必为此自责，因为 Gergely Kontra 和 Eric Van Dewoestine 也跟你差不多，只不过人家开发了 supertab.vim 这个插件，可以永远懒下去了，下面我来教你偷懒吧。

在你的 ~/.vimrc 文件中加上这两句：

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

以后当你准备按“Ctrl+X Ctrl+O”的时候直接按<Tab>就好了，够爽吧

我稍微再介绍一下上面那两句配置信息：

```
let g:SuperTabDefaultCompletionType="<C-X><C-O>"
" 设置按下<Tab>后默认的补全方式，默认是<C-P>，
" 现在改为<C-X><C-O>。关于<C-P>的补全方式，
" 还有其他的补全方式，你可以看看下面的一些帮助：
" :help ins-completion
" :help compl-omni
```

```
let g:SuperTabRetainCompletionType=2
" 0 - 不记录上次的补全方式
" 1 - 记住上次的补全方式，直到用其他的补全命令改变它
" 2 - 记住上次的补全方式，直到按 ESC 退出插入模式为止
```