vim使用进阶

使用vim提高你的文本编辑效率

Easwy Yang

vim使用进阶：使用vim提高你的文本编辑效率

Easwy Yang

目录

第 1 章 目录

本系列文章，是原来我在CSDN的Easwy专栏 [http://blog.csdn.net/easwy]撰写的"使用VIM开发软件项目"系列。Easwy的博客 [http://easwy.com/blog/]搬家以后，就把CSDN上的文章全部移到此处。

写本系列文章的最初想法，是介绍如何用vim开发软件。但纵观整个系列，讲述的其实和软件开发关系并不大，基本都在讲vim的使用技巧、vim的配置及vimrc、vim的命令和vim的插件。因此，把文章移到新站后，我把本系列的名字改为"vim使用进阶"，希望大家一如既往的支持该系列。

本文是这一系列文章的总目录，可由此访问本系列所有文章。

我使用docbook重新整理了一下这些文章，你可以在这里下载html版本的vim使用进阶2009年前文章打包 [http://easwy.com/blog/uploads/vim/advanced_vim_skills_html.zip]，现在也提供PDF版本 [http://easwy.com/blog/uploads/vim/advanced_vim_skills_pdf.zip]下载。

[目录]

本文介绍如何使用taglist插件(plugin)来查看程序中的标签(tag)，并介绍了taglist插件的配置和用法。taglist插件利用Exuberant ctags程序生成标签文件，并提供像Source Insight那样的标签窗口，可以方便的跳转到函数、变量等标签所在的位置。

- vim使用进阶： 文件浏览和缓冲区浏览 [http://easwy.com/blog/archives/advanced-vim-skills-netrw-bufexplorer-winmanager-plugin/]

  本节介绍了如何在vim中浏览文件，以及如何查看当前打开的缓冲区。利用netrw插件，可以方便的在vim中浏览各个目录、打开指定文件，而不用切换到文件浏览器或shell；当然，netrw插件的作用并不仅仅局限于此。利用bufexplorer插件，则可以方便的查看打开的缓冲区(buffer)，在缓冲区间进行切换。本文的最后介绍了winmanager插件，使用这个插件，可以把netrw插件、bufexplorer插件和taglist插件整合起来，使vim看起来更像一个集成开发环境(IDE)。

- vim使用进阶： lookupfile插件 [http://easwy.com/blog/archives/advanced-vim-skills-lookupfile-plugin/]

  如果你在开发一个大的项目，当你想在一大堆文件中查找或者编辑指定文件时，Lookupfile插件是必不可少的。使用它，可以快速查找项目文件、可以在缓冲区查找指定文件、可以浏览指定目录等。在查找时甚至可以使用正则表达式(regex)，在你只记得部分文件名或目录名时，这可是救命的手段。

- vim使用进阶: 开启文件类型检测 [http://easwy.com/blog/archives/advanced-vim-skills-filetype-on/]

  vim最吸引人的一点是，它支持无穷多的文件类型，而且能够随意扩展。在本文主要介绍如何打开文件类型检测的功能，以及如何使用基于文件类型的插件(filetype plugin)。正因为有了文件类型检测的功能，我们才可能针对不同的类型的文件，定义不同的键映射(map)、设置不同的选项，进行语法高亮的染色(后续文章中介绍)...你可以实现任意你想实现的功能。

- vim使用进阶: 乱花渐欲迷人眼 - 语法高亮 [http://easwy.com/blog/archives/advanced-vim-skills-syntax-on-colorscheme/]

  VIM并不是只有黑色两色。正相反，它提供了非常灵活的机制允许用户自定义色彩。运行在终端中的VIM，由于终端本身的限制，只能使用若干种固定的颜色；但对于GVIM来讲，你可以根据你的喜好调出任意的颜色。本文介绍的，正是vim的语法高亮功能。

- vim使用进阶: 程序员的利器 - cscope [http://easwy.com/blog/archives/advanced-vim-skills-cscope/]

  在前面的文章中介绍了利用ctags生成的tag文件，跳转到标签定义的地方。但如果想查找函数在哪里被调用，或者标签在哪些地方出现过，ctags就无能为力了，这时需要使用更为强大的cscope。本文就介绍如何使用cscope，有了它，你可以把source insight抛到一边去了。

- vim使用进阶: 剑不离手 - quickfix [http://easwy.com/blog/archives/advanced-vim-skills-quickfix-mode/]

  vim由一个程序员开发，并且为更多的程序员所使用，所以vim对开发人员的强大支持，也就可以理解了。quickfix模式的引入就是一个例子。quickfix模式，是一种加速你开发的工作方式，使你不必离开vim，就可以快速的完成"编辑-编译-修正"你的程序。它不仅仅对开发人员有用，只要你的工作有类似"编辑-编译-修正"的过程，它就可以极大的简化你的工作。

- vim使用进阶： 智能补全 [http://easwy.com/blog/archives/advanced-vim-skills-omin-complete/]

使用过Source Insight的人一定对它的自动补全功能印象深刻，在很多的集成开发环境中，也都支持自动补全。vim做为一个出色的编辑器，这样的功能当然少不了。本文主要介绍vim的OMNI补全。将在下一篇中介绍其它的补全方式。

- vim使用进阶: 自动补全 [http://easwy.com/blog/archives/advanced-vim-skills-auto-complete/]

  本文继续介绍vim的补全功能。作为一个通用的编辑器，vim实现的补全功能并不仅仅限于对程序的补全，它可以对文件名补全、根据字典进行补全、根据本缓冲区或其它缓冲区类似的内容进行补全、根据文件语法补全等等，它甚至允许用户自己编写函数来实现定制的补全。本文简要介绍了这些补全的方法。

- vim使用进阶: 指随意动，移动如飞（一）[http://easwy.com/blog/archives/advanced-vim-skills-basic-move-method/]

  VIM提供的移动方式多如牛毛，但我们并不需要掌握全部这些命令，只需要掌握最适合自己的那些命令。本文介绍了最常用的一些移动命令，在下篇文章中将介绍更高级的移动方法。

- vim使用进阶: 指随意动，移动如飞（二）[http://easwy.com/blog/archives/advanced-vim-skills-advanced-move-method/]

  本文介绍如何在vim中移动，主要涉及如何使用跳转表(jump-motions)、使用标记(mark)、使用折行(fold)，以及如何在程序中移动。

- vim使用进阶: 在VIM中使用GDB调试 [http://easwy.com/blog/archives/advanced-vim-skills-vim-gdb-vimgdb/]

  本文介绍了如何使用vimgdb补丁在vim中用gdb调试程序，同时还介绍了vim的编译方法。然而，vim只是一个编辑器，而不是一个集成开发环境(IDE)，所以它对调试的支持很有限。

- vim使用进阶: vim编译中遇到的问题及解决方法 [http://easwy.com/blog/archives/advanced-vim-skills-solve-compile-problem/]

  本文主要介绍vim编译中遇到的问题及解决办法。文中介绍了如何通过看config.log，来了解配置失败的原因。有些网友编译图形化的gvim失败，可以参照本文的方法解决。

- vim使用进阶: 在vim中使用拷贝/粘贴
  未完待续

- vim使用进阶: 映射自己的vim按键
  未完待续

- vim使用进阶: 巧用vim的缩写功能
  未完待续

- vim使用进阶: 强大的自动命令(autocmd)
  未完待续

- vim使用进阶: 在vim中使用gdb调试（二）
  未完待续，介绍clewn

- vim使用进阶: 在vim中使用gdb调试（三）
  未完待续，介绍pyclewn

- 其它你感兴趣的话题
  未完待续

第 2 章 序言

<< 返回vim使用进阶: 目录

从初次接触vim，到现在已经有好些年时间了。在软件开发中使用vim，不过是近两年的事情。对vim的了解远远算不上深入，不过还是把自己使用vim的一些经验写出来，希望对vim用户有所帮助。

本系列文章介绍我自己使用vim的一些经验，主要包括vim使用技巧、vim配置、vim命令、vim插件等内容。本篇是序言，务虚为主。

在使用vim进行软件开发之前，我使用的工具是Source Insight，相信大家并不陌生。Source Insight是一个不错的工具，特别是在浏览代码方面。在面对成百上千个文件组成的陌生源代码时，使用Source Insight可以让你很快的了解软件的主体流程、调用关系、类型定义.....。使用Source Insight写代码也不错，它的自动补全功能很强，似乎使用拷贝、粘贴，加上自动补全，就可以完成代码了。不再使用Source Insight的原因有二个，一是Source Insight只支持windows，不能在Linux上用；二是不想再用盗版的Source Insight。

在决定使用vim前，也在vim和emacs犹豫了很久，最终懒惰的天性让我放弃了emacs：实在是不习惯在移动光标时，也要用两只手按住CTRL/ALT/SHIFT再加个什么键；我的天性喜欢偷懒，能用一个手指做的事，不想用两个手指完成。所以最终选择了使用vim。现在感觉，vim和emacs在文本编辑方面不分伯仲。emacs的优势，在于它的可扩展性，使它可以完成很多和文本编辑无关的事情；不过这也造成了它的过于庞大。

刚改用vim，最不适应的是不能再用鼠标指哪儿打哪儿了（vim当然支持鼠标，只是双手在键盘、鼠标间切换，很是影响效率），然后是拷贝、粘贴，然后是查看调用关系，函数间跳转，再然后.....总之，刚开始的一段时间非常痛苦，效率也非常的低，甚至总是想着放弃。经过这段时间以后，逐渐摸到一些窍门，也渐渐的知道到哪里去寻求帮助，在哪儿可以找到vim的资源，工作效率一点点提升了上来。到现在，工作中已经离不开vim了，vim成了计算机中缺省的文本编辑工具！

其实，对vim和emacs这样的工具来说，它们最强大之处，在于它们的可定制性。由于它们的可定制性，你完全可以定制出一个符合你自己编辑习惯的编辑器，在这样一个编辑器里，你的工作效率将达到最高。当然，要达到这样的境界，你需要付出非常坚苦的努力！如果你的工作是以文本编辑为主，例如，你是一个程序员，那么付出这种努力是值得的，也是有回报的。如果你没有很多文本编辑工作要做，那么也没有必要耗费这么大的力气，来学习这些工具。

最后对vim的初学者提一些建议：

1. 如果你的工作以文本编辑(不是指Microsoft word中的文本编辑)为主，那么学习vim或emacs是值得的；

2. 刚开始使用vim或emacs的经历是很痛苦的，因为它们可能完成不同于你已经习惯的windows编辑器。我的建议是：坚持下去！咬牙坚持下去！你会获得回报的。

3. vim手册 (help files) 是学习如何高效使用vim的重要资源，一定要多读手册。如果你是因为害怕读英文手册而不肯学习vim的话，那么，到vim中文文档下载vim 7.0的中文手册，安装好后，再使用":help @cn"命令，你就可以看到中文手册了。

4. 如果你是第一次接触vim，那么使用":help tutor"或":help tutor@cn"，你就会看到一个30分钟的vim教程，会教会你vim的一些基本命令。

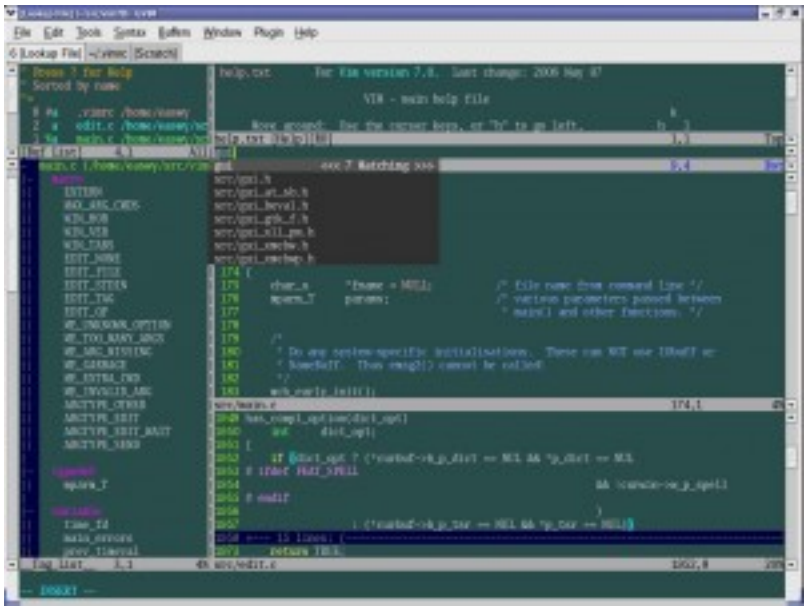
5. 有时间的话，一定要阅读一下Bram Moolenaar(vim的作者)写的Seven habits of effective text editing（七个有效的文本编辑习惯），你可以知道怎样提高你的编辑效率。在http://vimcdoc.sourceforge.net/下载PDF格式的手册，在附录二你可以看到它的中文译本。

本系列文章都针对vim 7.0版本，如果这里介绍的功能在你的vim中不存在，不妨升级到vim 7.0后再试一次。

我以Linux版本的vim 7.0为例。vim也有支持windows的版本，可以到vim主页下载预编译的的windows版vim 7.0。

文章尽量不涉及具体的软件源代码，如果确实需要，则以vim 7.0的源代码为例。此代码可以也可以在vim主页下载，我把它解压在~/src/vim70目录下。后续文章以此目录为例进行讲解。

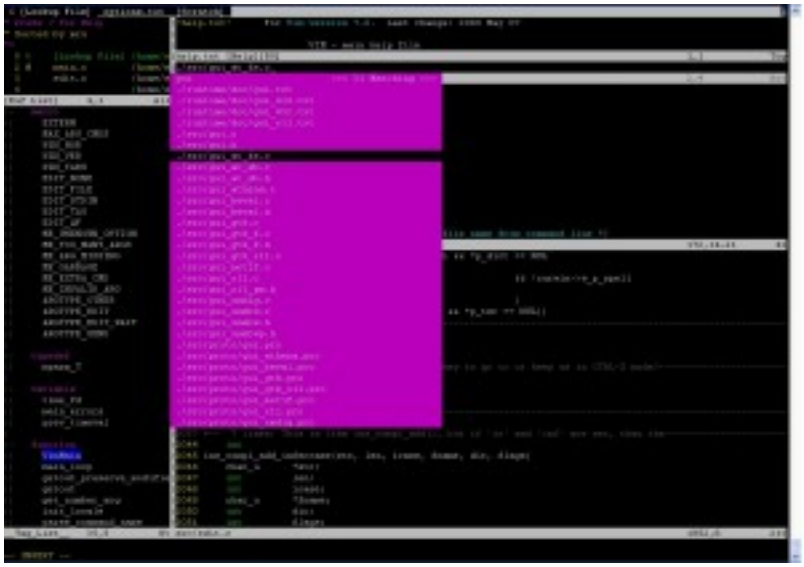
附图是我所使用vim环境，在这副图中，我打开了三个标签页，一个是主编程区，一个是打开的.vimrc文件，另外一个是草稿区。在主编程标签页中有几个不同的窗口，分别是当前打开的buffer，当前文件中的tag，help窗口，lookupfile窗口，src/main.c和src/edit.c。其中使用了三个vim插件（plugin）：winmanager, taglist, lookupfile。



点击查看大图

从这副图中可以看出vim的一些特性：多标签支持(tab)、多窗口支持、插件支持(plugin)、语法高亮功能(syntax)、文本折叠功能(folding).....这些特性，以及其它数不清的特性，我都将在后续文章中尽量给予介绍。

上图是GUI界面的vim，下图是控制台(console)下的vim的抓图。这是我通常的开发方式：使用putty远程登录到linux服务器上，使用控制台的vim开发软件：



点击查看大图

对vim的了解有限，请多指教。

<< 返回vim使用进阶: 目录

第 3 章　使用会话(session)和viminfo

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

```
:help mksession
:help 'sessionoptions'
:help source
:help wviminfo
:help rviminfo
:help 'viminfo'
```

很多软件都具有这样一种功能：在你下一次启动该软件时，它会自动为你恢复到你上次退出的环境，恢复窗口布局、所打开的文件，甚至是上次的设置。

那么，vim有没有这种功能呢？

答案当然是肯定的！这需要使用vim的会话(session)及viminfo的保存和恢复功能。

使用会话(session)和viminfo，可以把你编辑环境保存下来，然后你在下次启动vim后，可以再恢复回这个环境。我们在开发项目或书写文档时，其周期不是一两天。如果你在中途退出了vim而不能恢复原先的编辑环境的话，你又要重新打开你所打开的文件，重新定义你的映射(map)、缩写(abbreviate)，重新定位你所设定的标记的位置(mark)，重新设置项目相关设置(options).....不是一般的麻烦！

要恢复上次的编辑环境，我们需要保存两种不同的信息，一种是会话(session)信息，另外一种是viminfo信息。

- 会话信息中保存了所有窗口的视图，外加全局设置。
- viminfo信息中保存了命令行历史(history)、搜索字符串历史(search)、输入行历史、非空的寄存器内容(register)、文件的位置标记(mark)、最近搜索/替换的模式、缓冲区列表、全局变量等信息。

我们在下面分别对其进行介绍。

[会话]

我们可以使用:mksession [file]命令来创建一个会话文件，如果省略文件名的话，会自动创建一个名为Session.vim的会话文件。会话文件，其本质上是一个vim脚本，你可以使用上述命令生成一个session文件，然后再查看其中的内容，就会对session文件有一个深入的认识。

会话文件中保存哪些信息，是由'sessionoptions'选项决定的。缺省的'sessionoptions'选项包括: "blank,buffers,curdir,folds,help,options,tabpages,winsize"， 也就是会话文件会恢复当前编辑环境的空窗口、所有的缓冲区、当前目录、折叠(fold)相关的信息、帮助窗口、所有的选项和映射、所有的标签页(tab)、窗口大小。

如果你使用windows上的vim，并且希望你的会话文件可以同时被windows版本的vim和UNIX版本的vim共同使用的话，在'sessionoptions'中加入'slash'和'unix'，前者把文件名中的'\'替换为'/'，后者会把会话文件的换行符保存成unix格式。

如果你不希望在session文件中保存当前路径，而是希望session文件所在的目录自动成为当前工作目录，那么，需要在'sessionoptions'去掉'curdir',加入'sesdir'，这样每次载入session件时，此文件所在的目录就被设为vim的当前工作目录。在你通过网络访问其它项目的session文件

时，或者你的项目有多个不同版本（位于不同的目录），而你想始终使用一个session文件时，这个选项比较有用：你只需要把session文件拷贝到不同的目录，然后使用就可以了。设置此选项后，session文件中保存的是文件的相对路径，而不是绝对路径。

我们在上面使用:mksession命令创建了会话文件，那么怎么使用会话文件恢复编辑环境呢？很简单，你只需要使用:source session-file来导入会话文件。因为会话文件是一个脚本，里面保存的是Ex命令，所以"source"命令只是把会话文件中的Ex命令执行一遍。

[viminfo]

使用:wviminfo [file]命令，可以手动创建一个viminfo文件。

其实，在vim退出时，每次都会保存一个.viminfo文件在用户的主目录。我们使用:wviminfo命令则是手动创建一个viminfo文件，因为缺省的.viminfo文件会在每次退出vim时自动更新，谁知道你在关闭当前软件项目后，又使用vim做过些什么呢？这样的话，.viminfo中的信息，也许就与你所进行的软件项目无关了。还是手动保存一个保险。

":wviminfo"命令保存哪些内容，以及保存的数量，由'viminfo'选项决定，这个选项的值在windows上和在linux上的缺省值不同，具体含义参阅手册。

要读入你所保存的viminfo文件，使用:rviminfo [file]命令。

现在，回到我们的例子，依旧是上篇文章中的抓图，先看一下我们当前目录，执行:pwd，显示"/home/easwy/src/vim70"，接下来，执行下面的命令：

```
:cd src                             "切换到/home/easwy/src/vim70/src目录
:set sessionoptions-=curdir         "在session option中去掉curdir
:set sessionoptions+=sesdir         "在session option中加入sesdir
:mksession vim70.vim                "创建一个会话文件
:wviminfo vim70.viminfo             "创建一个viminfo文件
:qa                                 "退出vim
```

退出vim后，在命令行下执行gvim &，再次进入vim，这时看到的是一个空白窗口。然后执行下面的命令：

```
:source ~/src/vim70/src/vim70.vim  '载入会话文件
:rviminfo vim70.viminfo            '读入viminfo文件
```

太棒了，又恢复到昨天退出时的状态了！继续工作~~~~

不过，每次都要手工修改'sessionoptions'或'viminfo'吗？多麻烦啊.....别着急，现在是时候介绍vimrc了，请移步下一章: vimrc初步。

[参考文档]

1. vim手册

2. vim中文手册

<< 返回vim使用进阶: 目录

第 4 章 vimrc初步

<< 返回vim使用进阶: 目录

上一章我们介绍了会话(session)文件和viminfo文件，其中'sessionoptions'选项和'viminfo'选项的配置可能会根据你的需要进行调整。但如何保存你所做的调整呢？我们将在这一章中介绍。

本节所用命令的帮助入口：

```
:help compatible
:help mapleader
:help map
:help autocmd
```

为什么我的vim这么难用？不能语法高亮，没有折行，不能打开多个窗口多个buffer，不能.....

为什么别人用几个键就可以完成一个很复杂的功能，而我不能？

为什么别人的vim看起来和我的很不一样？

.....

当你开始问这些问题的时候，是时候去检查一下你的vimrc了。

当vim在启动时，如果没有找到vimrc或gvimrc，它缺省工作VI兼容的模式。这意味着，你只能使用VI所具备的功能，而vim中的大量扩展功能将无法使用。也许这就是你的vim如此难用的原因。

vim中自带了一个vimrc例子，让我们从这个例子开始吧。

下面我以Linux下的vim为例，windows版本的vim，会在后面提到。

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。

首先把这个示例的vimrc拷贝到相应的目录，在Linux下，应该把它拷贝到你的home目录下，名字为".vimrc"，下面是shell命令：

```
cp /usr/share/vim/vim70/vimrc_example.vim ~/.vimrc
```

或者你在vim中执行下面的命令，和上面的shell命令完成相同的功能：

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

现在，你退出vim后再进入，你的vim和刚才已经不一样了。

你可以先读一下你的vimrc，看看它都设定了什么：

```
:e ~/.vimrc
```

这是一个注释良好的文件，不需要多解释。

对windows版本的vim，它已经缺省的有了一个vimrc，你可以在vim在使用下面的命令来查看它：

:e $vim/_vimrc

在这个文件中，它包含了上面提到的vimrc_example.vim。同时，它会把vim设置的更符合windows的操作习惯。比如，支持CTRL-C拷贝，CTRL-V粘贴等等。Windows下的用户，可以使用$vim/_vimrc来做为你的第一个vimrc。

顺便提一句，在unix/linux中，文件名可以以"."开头，表明此文件是隐藏的。而在windows中，不允许文件名以"."开头，所以，windows版本的vim，将读取_vimrc来做为自己的配置文件。

在今后使用vim的日子里，你会频繁的更改你的vimrc。所以我们需要设置一些快捷方式，使我们能快速的访问vimrc。

把下面这段内容拷贝到你的vimrc中：

```
1    "Set mapleader
2    let mapleader = ","
3
4    "Fast reloading of the .vimrc
5    map <silent> <leader>ss :source ~/.vimrc<cr>
6    "Fast editing of .vimrc
7    map <silent> <leader>ee :e ~/.vimrc<cr>
8    "When .vimrc is edited, reload it
9    autocmd! bufwritepost .vimrc source ~/.vimrc
```

为了方便解释，我给每一行都加了一个行号。

- 在vimrc中，双引号开头的行，将被当作注释忽略。
- 第2行，用来设置mapleader变量，当mapleader为未设置或为空时，使用缺省的"\"来作为mapleader。
  mapleader变量是作用是什么呢？看下面的介绍。
- 第5行定义了一个映射(map)，这个映射把<leader>ss，映射为命令:source ~/.vimrc<cr>。当定义一个映射时，可以使用<leader>前缀。而在映射生效时，vim会把<leader>替换成mapleader变量的值。也就是说，我们这里定义的<leader>ss在使用时就变成了",ss"，当输入这一快捷方式时，就会source一次~/.vimrc文件(也就是重新执行一遍.vimrc文件)。
- 第7行，定义了<leader>ee快捷键，当输入,ee时，会打开~/.vimrc进行编辑。
- 第9行，定义了一个自动命令，每次写入.vimrc后，都会执行这个自动命令，source一次~/.vimrc文件。

有了上面的快捷键，我们就能快速的打开vimrc文件编辑，快速重新source vimrc文件，方便多了。

无论在windows还是在linux中，我都使用vim作为我的缺省编辑器。并且，我想使用同一个vimrc文件。因此，我定义了一个MySys()函数，用来区分不同的平台，以进行不同的配置。

另外，在编辑vimrc文件时，我更喜欢新开一个标签页来编辑，而不是在当前窗口中。因此，我定义了SwitchToBuf()函数，它在所有标签页的窗口中查找指定的文件名，如果找到这样一个窗口，就跳到此窗口中；否则，它新建一个标签页来打开vimrc文件。(注：标签页(tab)功能只有在vim 7.0版本以上才支持。)

下面是我的vimrc中使用的设定，希望能够抛砖引玉：

```
" Platform
function! MySys()
  if has("win32")
    return "windows"
  else
    return "linux"
  endif
endfunction

function! SwitchToBuf(filename)
    "let fullfn = substitute(a:filename, "^\\~/", $HOME . "/", "")
    " find in current tab
    let bufwinnr = bufwinnr(a:filename)
    if bufwinnr != -1
        exec bufwinnr . "wincmd w"
        return
    else
        " find in each tab
        tabfirst
        let tab = 1
        while tab <= tabpagenr("$")
            let bufwinnr = bufwinnr(a:filename)
            if bufwinnr != -1
                exec "normal " . tab . "gt"
                exec bufwinnr . "wincmd w"
                return
            endif
            tabnext
            let tab = tab + 1
        endwhile
        " not exist, new tab
        exec "tabnew " . a:filename
    endif
endfunction

"Fast edit vimrc
if MySys() == 'linux'
    "Fast reloading of the .vimrc
    map <silent> <leader>ss :source ~/.vimrc<cr>
    "Fast editing of .vimrc
    map <silent> <leader>ee :call SwitchToBuf("~/.vimrc")<cr>
    "When .vimrc is edited, reload it
    autocmd! bufwritepost .vimrc source ~/.vimrc
elseif MySys() == 'windows'
    " Set helplang
    set helplang=cn
    "Fast reloading of the _vimrc
    map <silent> <leader>ss :source ~/_vimrc<cr>
    "Fast editing of _vimrc
    map <silent> <leader>ee :call SwitchToBuf("~/_vimrc")<cr>
    "When _vimrc is edited, reload it
    autocmd! bufwritepost _vimrc source ~/_vimrc
endif
```

```
" For windows version
if MySys() == 'windows'
    source $VIMRUNTIME/mswin.vim
    behave mswin
endif
```

注意：我在windows中也定义一个"HOME"环境变量，然后把_vimrc放在"HOME"环境变量所指向的目录中。如果你打算在windows中使用上面的设定，也需要这样做！

好了，现在我们知道如何永久更改'sessionoptions'选项和'viminfo'选项了，把对它们的配置放入你的vimrc即可。

vim自带的示例vimrc中，只定义最基本的配置。

在http://www.amix.dk/vim/vimrc.html有一个非常强大的vimrc，有人戏称为"史上最强的vimrc"，或许有些言过其实。不过，如果你通读了这个vimrc，相信能从中学到很多.....

这里有一个redguardtoo修改过的版本，可以对照参考一下。

我的vimrc也是基于Amix的模板，然后自己做了很多修改。

建议：不要照拷这个vimrc！可能这个文件的设定并不符合你的习惯。另外，这个文件的设定，可能也不能在你的工作环境中运行。

[参考文档]

- vim手册
- vim中文手册
- http://www.amix.dk/vim/vimrc.html

<< 返回vim使用进阶: 目录

第 5 章 保存项目相关配置

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

```
:help 'path'
:help mksession
:help find
:help gf
:help CTRL-W_f
```

我们在前面介绍了vimrc，vimrc定义了vim通常的行为。然而，每个项目都有其特殊的定义，虽然我们也可以在vimrc中对每个项目进行定制，但这样一来，vimrc会变得很大，使vim启动速度变慢；另外也会使vimrc变得难以维护。

因此，我们使用其它的方法来保存项目相关的信息，这就是本文的主要内容。我们将以path选项的设置为例进行讲解。

path选项定义了一个目录列表，在使用gf，find，以及CTRL-W f等vim命令时，如果使用的是相对路径，那么就会在path选项定义的目录列表中查找相应的文件。path选项以逗号分隔各目录名。我们依旧以vim 7.0的源代码为例（源代码放在~/src/vim70/目录中）。

对于这个项目，我们的path选项设置如下：

```
set path=.,/usr/include,,~/src/vim70/**
```

稍微解释一下各项的含义，更详细的信息，请查看path选项的帮助页：

- . 在当前文件所在目录中搜索
- /usr/include 在/usr/include目录中搜索
- ,, 在当前工作路径中搜索
- ~/src/vim70/** 在~/src/vim70的所有子目录树中进行搜索

设置了path选项后，怎么用呢？

我们还是回到序言中的屏幕抓图，我们把光标定位到src/main.c文件第22行的"fcntl.h"单词上，然后在Normal模式下按"gf"。咦，vim打开了/usr/include/fcntl.h文件！

现在我们按"CTRL-^"回到刚才的位置，光标仍旧定位在第22行的"fcntl.h"单词上，然后按"CTRL-W f"。啊哈，这次vim打开了一个水平分隔窗口，在此窗口中打开了/usr/include/fcntl.h。

尽管在src/main.c中未指定fcntl.h的路径，但vim会在path选项定义的路径中搜索此文件，方便吧！

现在我们看一下"find"命令，输入：

```
:find netrw.vim
```

vim打开了~/src/vim70/runtime/autoload/netrw.vim文件。用这种方法打开文件真是太方便了，你不用输入文件的路径，vim会自动在path选项定义的路径中搜索。不过"find"命令也有缺陷，如果你只记得文件名的一部分，那么就没有办法用find命令打开这个文件了。而且find命令也不允许使用正则表达式。没关系，我们还有更好的方法来打开文件，我将在Lookupfile插件中介绍这些方法。

path选项介绍完了，我们进入正题，如何把本项目相关的配置保存起来，下次打开本项目时自动恢复这些配置呢？

我们有两种方法做到这一点。

[方法1]

我们在~/src/vim70/目录下建立一个文件，假定文件名为workspace.vim，文件内容为：

```
"set project path
set path+=~/src/vim70/**
```

这个文件中保存了项目相关的信息，例如选项值，键映射，函数定义，自动命令，等等。我们的例子中只定义的path选项，我们没有使用"set path=..."语句，在vim手册中建议使用"set path+=..."和"set path-=..."格式。

接下来，在你的vimrc文件中加入下面的语句：

```
" execute project related configuration in current directory
if filereadable("workspace.vim")
    source workspace.vim
endif
```

以后，每次你在~/src/vim70/目录中启动vim时，vim都会自动载入workspace.vim，恢复项目的配置信息。

[方法2]

还记得本系列文章的第二篇使用会话(session)和viminfo吗？那篇文章中，我们介绍了使用session文件和viminfo保存项目环境的方法。如果你使用了会话文件，那么选项值，键映射，和其它很多信息都已经保存了。但会话的功能毕竟有限，不能把项目相关的配置全部保存下来，怎么办呢？

vim的作者已经想到了这个问题，并提供了解决办法。

在vim载入会话文件的最后一步，它会查找一个额外的文件并执行其中的ex命令。查找的规则是，把会话文件名的后缀去掉，然后在后面加上"x.vim"，。假设你的会话文件名为example.session，vim就会查找是否有examplex.vim，如果找到，就会执行此文件中的ex命令。

好了，我们先创建我们的会话文件：

```
:cd ~/src/vim70
:set sessionoptions-=curdir        '在session option中去掉curdir
:set sessionoptions+=sesdir        '在session option中加入sesdir
:mksession vim70s.vim              '创建一个会话文件
```

然后再编辑一个名为~/src/vim70/vim70sx.vim的文件，文件的内容为（当然，你可以在这个文件中加入更多内容）：

```
"set project path
set path+=~/src/vim70/**
```

退出vim后，在命令行下执行"gvim &"，再次进入vim，这时看到的是一个空白窗口。然后执行下面的命令：

```
:source ~/src/vim70/vim70s.vim  '载入会话文件
```

太棒了！原来的会话环境已经恢复，并且项目相关的配置也设置好了！

[参考文档]

1. vim手册

2. vim中文手册

<< 返回vim使用进阶: 目录

第 6 章 使用标签(tag)文件

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

```
:help 'tags'
:help :tag
:help :tags
:help CTRL-]
:help CTRL-T
:help vimgrep
:help cw
:help pattern
```

尽管相关的文章已经很多了，但tag文件实在是太有用了，所以还是啰嗦一次。

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。

对于程序来说，Tag文件中保存了诸如函数、类、结构、宏等的名字，它们所处的文件，以及如何通过Ex命令跳转到这些标签。它是一个纯文本文件，因此你可以手工的编辑它，也可以使用脚本对其进行操作。

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(在大多数Linux系统上，它是缺省的ctags程序)，它能够支持多达33种程序语言，足以满足我们开发的需要了。这里是它的中文手册。如果你的系统上未安装此程序，请到http://ctags.sourceforge.net下载。

emacs则使用etags来生成tag文件，如果希望vim也能支持etags的tag文件格式，需要编译vim时加入"+emacs_tags"选项。在这篇文章介绍了编译vim的方法。

Tag文件需要遵循一定的格式，由Exuberant Ctags生成的tag文件，缺省是如下格式：

```
{tagname} {TAB} {tagfile} {TAB} {tagaddress} {term} {field} ..
```

- {tagname} - 标识符名字，例如函数名、类名、结构名、宏等。不能包含制表符。
- {tagfile} - 包含 {tagname} 的文件。它不能包含制表符。
- {tagaddress} - 可以定位到 {tagname}光标位置的 Ex 命令。通常只包含行号或搜索命令。出于安全的考虑，vim会限制其中某些命令的执行。
- {term} - 设为 ;" ，这是为了兼容Vi编辑器，使Vi忽略后面的{field}字段。
- {field} .. - 此字段可选，通常用于表示此{tagname}的类型是函数、类、宏或是其它。

在{tagname}、{tagfile}和{tagaddress}之间，采用制表符(TAB符，即C语言中的"\t")分隔，也就是说{tagname}、{tagfile}的内容中不能包含制表符。

Tag文件的开头可以包含以"!_TAG_"开头的行，用来在tag文件中加入其它信息。vim能够识别两种这样的标记，经常用到的是"_TAG_FILE_SORTED"标记，例如：

!_TAG_FILE_SORTED<Tab>1<Tab>{anything}

上面这个标记说明tag文件是经过排序的，并且排序时区分了大小写，对排序的tag，vim会使用二分法来进行查找，大大加快了查找速度；如果值为0，则表示tag文件未经排序；如果值为2，则表示tag文件是忽略大小写排序的。

之所以在这里介绍tag文件的格式，是因为我们在后面提到的lookupfile插件中，会自己生成tag文件。

虽然ctags有为数众多的选项，但通常我们所使用的非常简单。还是以vim 7.0的代码为例，我们执行：

```
cd ~/src/vim70
ctags  -R src
```

上面这条命令会在~/src/vim70/目录下生成一个名为tags的文件，这个文件中包含~/src/vim70/src/目录下所有.c、.h文件中的标签。它一个文本文件，你可以用vim打开它看一下。此文件缺省按区分字母大小写排序，所以直接可以被vim使用。

现在我们进入vim，执行下面的命令：

```
:cd ~/src/vim70 "切换当前目录为~/src/vim70
:set tags=tags "设置tags选项为当前目录下的tags文件
```

现在，我们设置好了tags选项，接下来我们使用它：

```
:tag VimMain
```

你会看到vim打开了src/main.c文件，并把光标定位到第167行VimMain上。

我们知道，一般主程序的函数名为main，如果你尝试下面的命令：

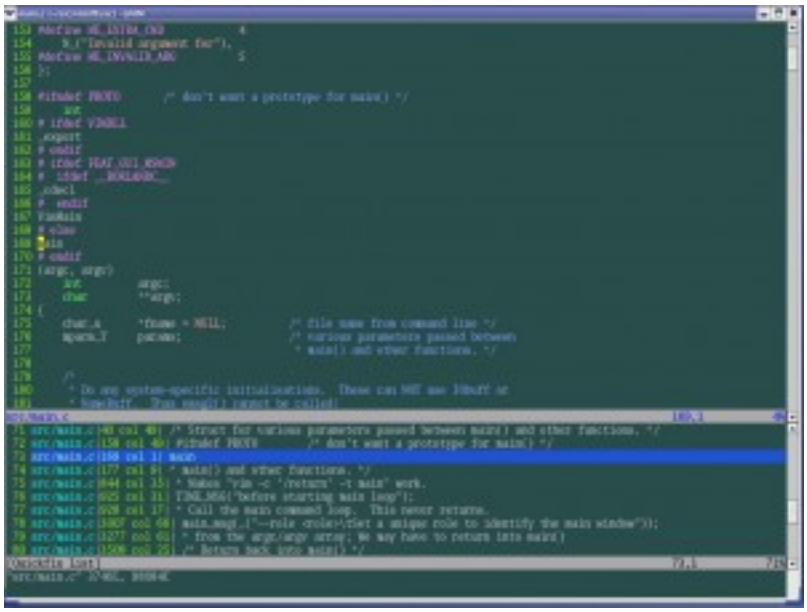
```
:tag main
# pri kind tag               file
1 F   f    main              src/xxd/xxd.c
main(argc, argv)
2 FS  d    main              src/if_python.c
46
Choice number (<Enter> cancels):
```

这里并没有src/main.c文件，怎么回事呢？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src/main.c中main()函数的标签。你可以在生成tag文件时给ctags指定参数来解决这个问题。见ctags手册。

或者你可以用":grep"或":vimgrep"来查找main(这篇文章讲解grep及vimgrep的用法)：

```
:cd ~/src/vim70
:vimgrep /\<main\>/ src/*.c
:cw
```

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src/main.c)，按回车，就可以跳到对应的位置了。



点击查看大图

如果你只记得部分的标签名，那么可以使用"tag"命令的搜索模式，你可以输入一个vim正则表达式来表示你所查找的名字，如：

```
:tag /\C\<\k\+ain\>
# pri kind tag               file
1 F   f    VimMain           src/main.c
VimMain
2 F   d    bindtextdomain    src/vim.h
483
3 F   d    bindtextdomain    src/vim.h
502
4 F   d    bindtextdomain    src/vim.h
504
5 F   f    main              src/xxd/xxd.c
main(argc, argv)
6 F   d    textdomain        src/vim.h
488
7 F   d    textdomain        src/vim.h
510
8 F   d    textdomain        src/vim.h
512
9 FS  d    bindtextdomain    src/gui_gtk.c
54
10 FS  d    bindtextdomain    src/gui_gtk_x11.c
37
11 FS  f    cmdsrv_main       src/main.c
cmdsrv_main(argc, argv, serverName_arg, serverStr)
12 FS  d    main              src/if_python.c
46
```

```
13 FS  d    textdomain        src/gui_gtk.c
51
14 FS  d    textdomain        src/gui_gtk_x11.c
34
Choice number (<Enter> cancels):
```

这表示我想查找一个以一个或多个keyword开始的标签，此标签以ain做为结尾，在查找时区分大小写。要读懂这个正则表达式，请":help pattern"。

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跳转命令见下(一般只需要知道CTRL-]和CTRL-T就可以了)：

```
:tag {ident}            "跳转到指定的标签
:tags                   "显示标签栈
CTRL-]                  "跳转到当前光标下的标签
CTRL-T                  "跳到标签栈中较早的标签
```

如果想了解更多命令，可以":help 29.1"(强烈建议程序员完整的阅读usr_29.txt和usr_30.txt)。

如果想更深入了解tag命令和相关知识，可以":help tagsrch"。

我之前写的一篇关于ctags和cscope的文章，参见：Vim + Cscope/Ctags

[参考文档]

- vim手册
- vim中文手册

<< 返回vim使用进阶: 目录

第 7 章 使用taglist插件

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

```
:help helptags
:help taglist.txt
```

上篇文章介绍了在vim中如何使用tag文件，本文主要介绍如何使用taglist插件(plugin)。

想必用过Source Insight的人都记得这样一个功能：SI能够把当前文件中的宏、全局变量、函数等tag显示在Symbol窗口，用鼠标点上述tag，就跳到该tag定义的位置；可以按字母序、该tag所属的类或scope，以及该tag在文件中出现的位置进行排序；如果切换到另外一个文件，Symbol窗口更新显示这个文件中的tag。

在vim中的taglist插件所实现的就是上述类似的功能，有些功能比SI弱，有些功能比SI更强。而且，taglist插件还在不断完善中！

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

在文章vimrc初步中，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；在上篇文章中，我们也已用到了Exuberant ctags；system()调用在一般的vim版本都会支持(suse Linux发行版中出于安全考虑，关闭了此功能)，所以我们已经满足了这三个条件。

现在我们到http://www.vim.org/scripts/script.php?script_id=273下载最新版本的taglist plugin，目前版本是4.3。

下载后，把该文件在~/.vim/目录中解压缩，这会在你的~/.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
```

注：windows用户需要把这个插件解压在你的$vim/vimfiles或$HOME/vimfiles目录。

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
```

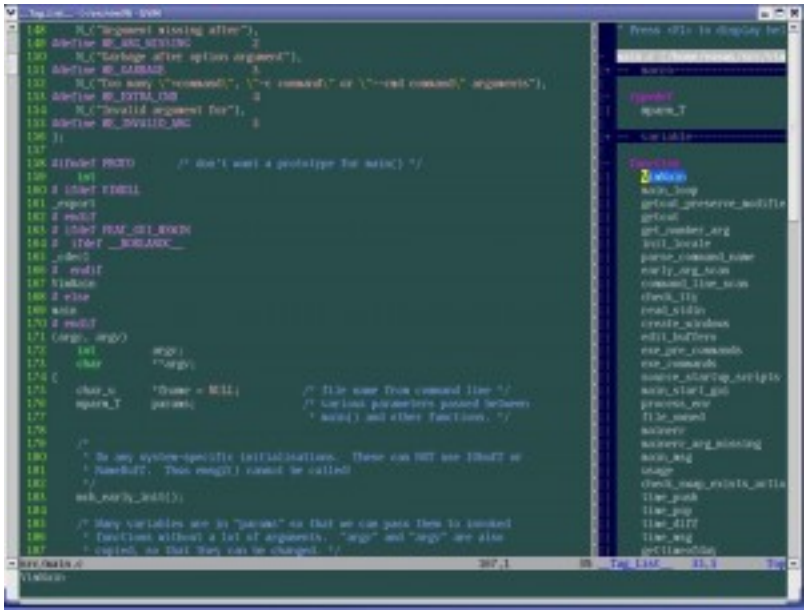
生成帮助标签后，你就可以用下面的命令查看taglist的帮助了：

```
:help taglist.txt
```

Taglist提供了相当多的功能，我的vimrc中这样配置：

```
"""""""""""""""""""""""""""""""
" Tag list (ctags)
"""""""""""""""""""""""""""""""
if MySys() == "windows"                 "设定windows系统中ctags程序的位置
let Tlist_Ctags_Cmd = 'ctags'
elseif MySys() == "linux"               "设定linux系统中ctags程序的位置
let Tlist_Ctags_Cmd = '/usr/bin/ctags'
endif
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
```

这样配置后，当你输入":TlistOpen"时，显示如下窗口：



点击查看大图

在屏幕右侧出现的就是taglist窗口，你从中可以看到在main.c文件中定义的所有tag：宏、定义、变量、函数等；你也可以双击某个tag，跳到该tag定义的位置；你还可以把某一类的tag折叠起来（使用了vim的折行功能），方便查看，就像上图中macro和variable那样。更多的功能，请查看taglist的帮助页，本文也会介绍一些常用功能。

下面介绍常用的taglist配置选项，你可以根据自己的习惯进行配置：

- Tlist_Ctags_Cmd选项用于指定你的Exuberant ctags程序的位置，如果它没在你PATH变量所定义的路径中，需要使用此选项设置一下；
- 如果你不想同时显示多个文件中的tag，设置Tlist_Show_One_File为1。缺省为显示多个文件中的tag；
- 设置Tlist_Sort_Type为"name"可以使taglist以tag名字进行排序，缺省是按tag在文件中出现的顺序进行排序。按tag出现的范围（即所属的namespace或class）排序，已经加入taglist的TODO List，但尚未支持；
- 如果你在想taglist窗口是最后一个窗口时退出vim，设置Tlist_Exit_OnlyWindow为1；
- 如果你想taglist窗口出现在右侧，设置Tlist_Use_Right_Window为1。缺省显示在左侧。

- 在gvim中，如果你想显示taglist菜单，设置Tlist_Show_Menu为1。你可以使用Tlist_Max_Submenu_Items和Tlist_Max_Tag_Length来控制菜单条目数和所显示tag名字的长度；
- 缺省情况下，在双击一个tag时，才会跳到该tag定义的位置，如果你想单击tag就跳转，设置Tlist_Use_SingleClick为1；
- 如果你想在启动vim后，自动打开taglist窗口，设置Tlist_Auto_Open为1；
- 如果你希望在选择了tag后自动关闭taglist窗口，设置Tlist_Close_On_Select为1；
- 当同时显示多个文件中的tag时，设置Tlist_File_Fold_Auto_Close为1，可使taglist只显示当前文件tag，其它文件的tag都被折叠起来。
- 在使用:TlistToggle打开taglist窗口时，如果希望输入焦点在taglist窗口中，设置Tlist_GainFocus_On_ToggleOpen为1；
- 如果希望taglist始终解析文件中的tag，不管taglist窗口有没有打开，设置Tlist_Process_File_Always为1；
- Tlist_WinHeight和Tlist_WinWidth可以设置taglist窗口的高度和宽度。Tlist_Use_Horiz_Window为1设置taglist窗口横向显示；

在taglist窗口中，可以使用下面的快捷键：

<CR>	跳到光标下tag所定义的位置，用鼠标双击此tag功能也一样
o	在一个新打开的窗口中显示光标下tag
<Space>	显示光标下tag的原型定义
u	更新taglist窗口中的tag
s	更改排序方式，在按名字排序和按出现顺序排序间切换
x	taglist窗口放大和缩小，方便查看较长的tag
+	打开一个折叠，同zo
-	将tag折叠起来，同zc
*	打开所有的折叠，同zR
=	将所有tag折叠起来，同zM
[[	跳到前一个文件
]]	跳到后一个文件
q	关闭taglist窗口
<F1>	显示帮助

可以用":TlistOpen"打开taglist窗口，用":TlistClose"关闭taglist窗口。或者使用":TlistToggle"在打开和关闭间切换。在我的vimrc中定义了下面的映射，使用<F9>键就可以打开/关闭taglist窗口：

```
map <silent> <F9> :TlistToogle<cr>
```

Taglist插件还提供了很多命令，你甚至可以用这些命令创建一个taglist的会话，然后在下次进入vim时加载此会话。

Taglist插件还可以与winmanager插件协同使用，这将在下篇文章中介绍。

[参考文档]

- vim手册

- vim中文手册
- taglist手册

<< 返回vim使用进阶: 目录

第 8 章 文件浏览和缓冲区浏览

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

```
:help netrw-browse
:help bufexplorer
:help winmanager
```

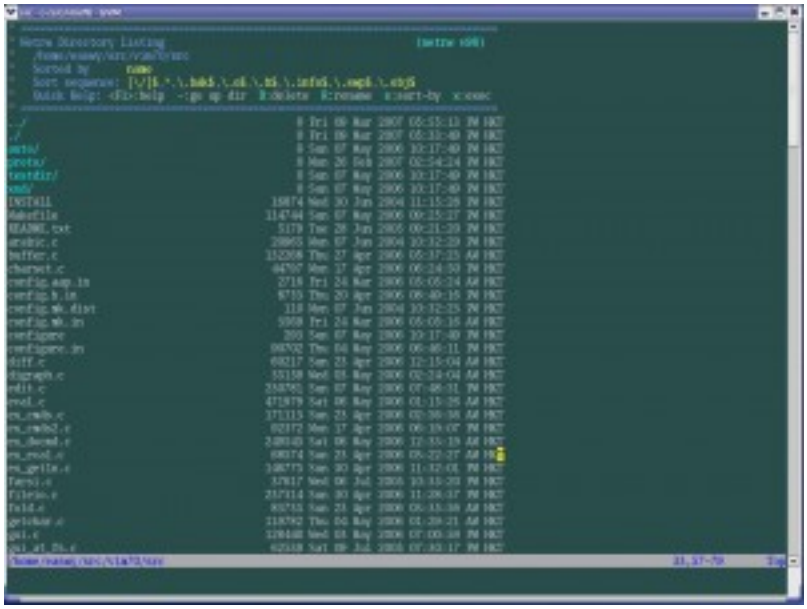
[文件浏览]

在开发过程中，经常需要查找某个文件。我们前面在介绍path选项时，介绍过使用find命令查找文件的方法。本节介绍vim的文件浏览插件。

在vim 7.0以前，文件浏览功能由explorer.vim插件提供，在vim 7.0中，这个插件被netrw.vim插件所代替。本文以vim 7.0为例，主要介绍netrw.vim插件。

netrw.vim是vim的标准插件，它已经伴随vim而发行，不需要安装。

我们现在试一下vim文件功能，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（从操作系统的视角来看，目录其实是一种特殊的文件）。例如，我们在vim中执行命令":e ~/src/vim70/src/"，会显示下面的窗口：



点击查看大图

这个窗口类似于文件管理器，你可以创建、删除、改名文件或目录；在目录上按回车时，会进入该目录；在文件上按回车时，会使用vim打开该文件；可以更改排序方式、排序风格；隐藏目录或文件（使之不在上述窗口中显示）等等。

Netrw插件中常用键绑定有：

```
<F1>        显示帮助
<cr>        如果光标下为目录，则进入该目录；如果光标下是文件，则用vim打开该文件
```

```
-           返回上级目录
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
i           切换显示方式
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
```

其它常用键，诸如使用书签、隐藏符合条件的文件等，请参阅netrw帮助页。

上面我们用":e ~/src/vim70/src/"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。我的vimrc中这样配置：

```
"""""""""""""""""""""""""""""""
" netrw setting
"""""""""""""""""""""""""""""""
let g:netrw_winsize = 30
nmap <silent> <leader>fe :Sexplore!<cr>
```

这样，在我输入",fe"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30。

浏览本地文件只是netrw插件的一项小功能，netrw插件最主要的功能是支持远程文件读写。利用该插件，你可以通过ftp，ssh，http等多种协议来编辑远程文件，也可以浏览远程机器的目录。

在软件开发过程中不常使用此功能，本文中不再介绍。参阅netrw手册页获取更多信息。

[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"Ex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，这会在你的~/.vim/plugin和~/.vim/doc目录中各放入一个文件：

```
plugin/ bufexplorer.vim - bufexplorer插件
doc/ bufexplorer.txt    - bufexplorer帮助文件
```

注：windows用户需要把这个插件解压在你的$vim/vimfiles或$HOME/vimfiles目录。

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
```

然后，就可以使用":help bufexplorer"命令查看BufExplorer的帮助文件了。

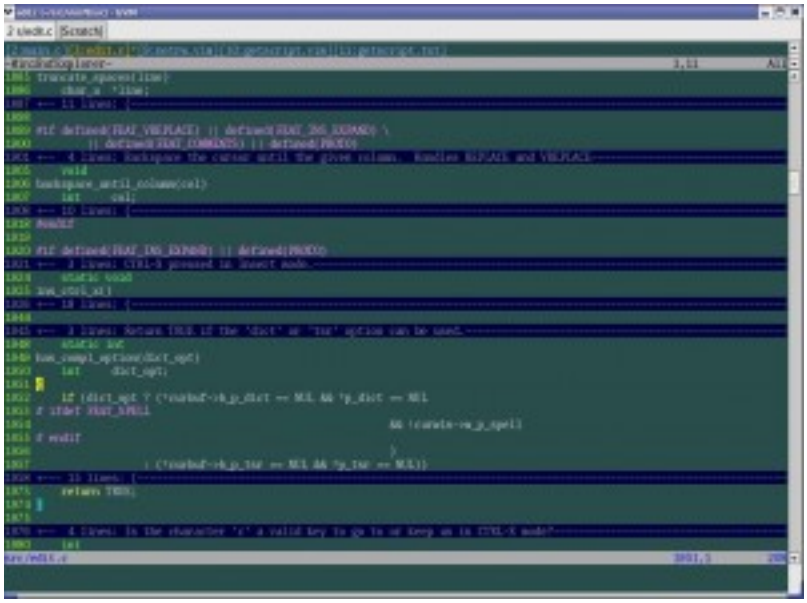
BufExplorer功能比较简单，这里就不做介绍了。我的vimrc里这样设置BufExplorer插件：

```
"""""""""""""""""""""""""""""""
" BufExplorer
"""""""""""""""""""""""""""""""
let g:bufExplorerDefaultHelp=0       " Do not show default help.
let g:bufExplorerShowRelativePath=1  " Show relative paths.
let g:bufExplorerSortBy='mru'        " Sort by most recently used.
let g:bufExplorerSplitRight=0        " Split left.
let g:bufExplorerSplitVertical=1     " Split vertically.
let g:bufExplorerSplitVertSize = 30  " Split width
let g:bufExplorerUseCurrentWindow=1  " Open in new window.
autocmd BufWinEnter \[Buf\ List\] setl nonumber
```

BufExplorer已经映射了几个键绑定，例如，使用",bv"就可以打开一个垂直分割窗口显示当前的缓冲区。

有一个称为minibufexpl.vim的插件，也可以把缓冲区列表显示出来，这个插件在此处下载：http://vim.sourceforge.net/scripts/script.php?script_id=159

使用这个插件后，屏幕截图看起来是这样，最上面一个窗口就是minibuffer窗口，列出了当前打开的缓冲区：



点击查看大图

这个插件没有帮助文件，参考下载页上的说明，以及脚本代码来进行配置。

[winmanager插件]

winmanager插件可以把上面介绍的Explorer插件(vim 7.0以前的文件浏览插件)和BufExplorer插件集成在一起，我们上篇文章中介绍过的taglist插件也提供了对winmanager插件的支持。

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，这会把winmanager插件解压到~/.vim/plugin和~/.vim/doc目录中：

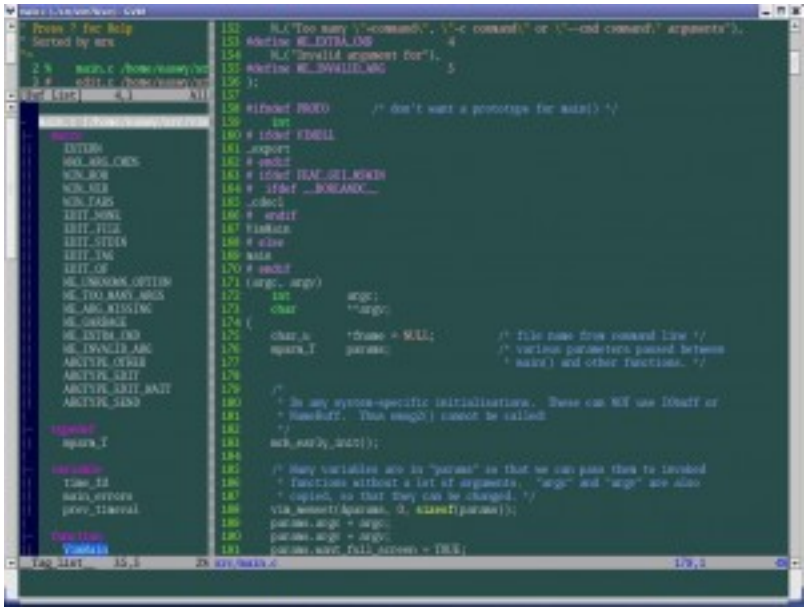
plugin/winmanager.vim - winmanager插件
plugin/winfileexplorer.vim - 改良的Explorer插件
plugin/wintagexplorer.vim - winmanager提供的tag插件，用处不大
doc/winmanager.txt - 帮助文件

仍然用":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。我的vimrc中这样设置：

```
"""""""""""""""""""""""""""""""
" winManager setting
"""""""""""""""""""""""""""""""
let g:winManagerWindowLayout = "BufExplorer,FileExplorer|TagList"
let g:winManagerWidth = 30
let g:defaultExplorer = 0
nmap <C-W><C-F> :FirstExplorerWindow<cr>
nmap <C-W><C-B> :BottomExplorerWindow<cr>
nmap <silent> <leader>wm :WMToggle<cr>
```

g:winManagerWindowLayout变量的值定义winmanager的窗口布局，使用上面的设置，我们的窗口布局看起来是这样的：



点击查看大图

左边有两个窗口，上面的是BufExplorer窗口，下面是taglist窗口。FileExplorer窗口和BufExplorer共用一个窗口，在BufExplorer窗口中使用"CTRL-N"可以切换到FileExplorer窗口，再使用一次则又切换回BufExplorer窗口。也就是说，在变量g:winManagerWindowLayout中，使用","分隔的插件，在同一个窗口中显示，使用"CTRL-N"在不同插件间切换；使用"|"分隔的插件，则在另外一个窗口中显示。

在屏幕右边的窗口则是主编辑区。

在上面的vimrc设置中，还定义了三个键映射，分别用于跳到左上窗口、左下窗口，以及显示/关闭winmanager窗口。

注：安装后，如果未设置g:winManagerWindowLayout变量，winmanager插件需要与BufExplorer插件一起才能使用。所以需要下载BufExplorer。

在http://vim.sourceforge.net/scripts/script.php?script_id=1440有一个winmanager插件的修改版本，如果在FileExplorer中打开文件时，它会使用与该文件相关联的程序来打开该文件，而不是使用vim。我没有使用过这个插件，有兴趣你可以试试。

[参考文档]

- vim帮助文件
- vim中文手册

<< 返回vim使用进阶: 目录

第 9 章 使用lookupfile插件

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

:help lookupfile

在文章保存项目相关配置中，我们讲过通过":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。

相比之下，在source insight中查找文件非常简单，只要输入部分的文件名，就可以找到符合条件的文件。

我一直被这个问题所困扰，直到有一天，在http://www.vim.org/上发现了一个非常出色的插件，才彻底解决了查找文件效率低下的问题，它的功能毫不逊于source insight。在给该插件投票时，我选择了"Life Changing"。是的，它改变了我的生活！

这个插件的名字是：lookupfile!

Lookupfile插件可以在下面的链接下载：http://www.vim.org/scripts/script.php?script_id=1581

它使用vim 7.0中插入模式下的下拉菜单补全功能，因此只能在vim 7.0及以上版本中使用。

下载该插件后，把它解压到你的~/.vim目录中，就完成了安装。然后在vim中执行":helptags ~/.vim/doc"命令，生成help文件索引，然后就可以使用":help lookupfile"命令查看lookupfile插件的帮助文件了。

注：windows用户需要把这个插件解压在你的$vim/vimfiles或$HOME/vimfiles目录。

Lookupfile插件还需要最新的genutils支持，因此，需要下载genutils：http://www.vim.org/scripts/script.php?script_id=197

这个插件提供了一些通用的函数，可供其它的脚本使用。它的安装方法也是在~/.vim目录解压就可以了。需要注意的是，最新版本的genutils使用了新的自动加载方式，所以和以前的版本不兼容。如果你的其它插件需要使用genutils的旧版本的话，你需要参照genutils的说明进行配置，以便使新旧两个版本能协同工作。

现在我们介绍lookupfile插件。虽然名字为lookupfile，其实这个插件它不仅能用来查找文件，还可以在打开的缓冲区中查找，按目录查找，等等。

[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。

我编写了下面的shell脚本，为vim70的源代码，生成一个文件名tag文件。

```
#!/bin/sh
# generate tag file for lookupfile plugin
echo -e "!_TAG_FILE_SORTED\t2\t/2=foldcase/" > filenametags
find . -not -regex '.*\.\(png\|gif\)' -type f -printf "%f\t%p\t1\n" | \
    sort -f >> filenametags
```

回想一下我们在"使用标签(tag)文件"一文中介绍的tag文件的格式。再对照脚本命令来看：

- echo命令用来生成filenametags文件中的"!_TAG_FILE_SORTED"行，表明此tag文件是经过排序的。
- find命令用来查找所有类型为普通文件，文件后缀名不是.png和.gif的文件，找到的文件按"文件名\t文件路径\t1"的格式输出出来。
- sort命令则把find命令的输出重新排序，然后写入filenametags文件中

在~/src/vim70/目录下运行该脚本，会生成一个名为filenametags的文件，包含了vim70目录下的所有文件的名字及其所在目录。

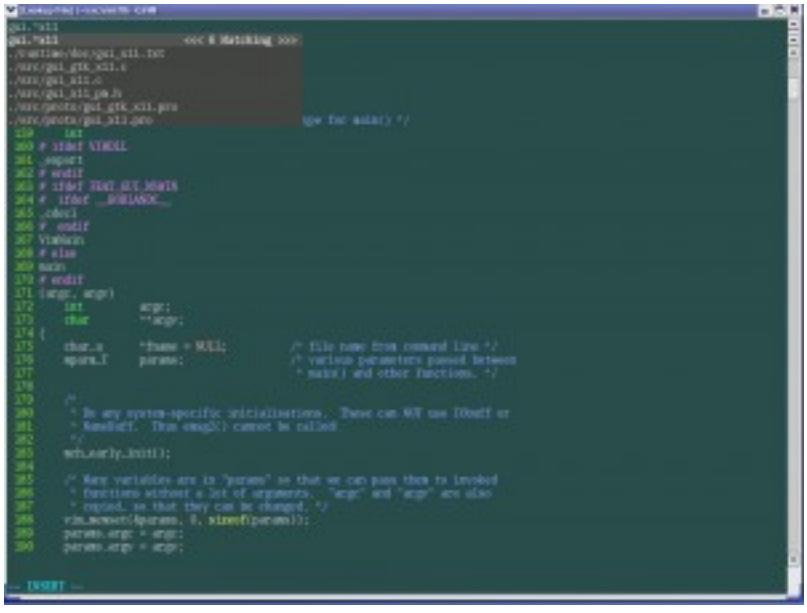
现在我们需要让lookupfile插件知道到哪去找文件名tag文件。我们假定vim当前工作目录为~/src/vim70/，执行下面的命令：

```
:let g:LookupFile_TagExpr = '"./filenametags"'
```

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了。文件名可以使用vim的正则表达式，这大大方便了文件的查找。你可以用"CTRL-N"和"CTRL-P"（或者用上、下光标键）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。

下图是使用lookupfile插件查找文件的一个例子：



点击查看大图

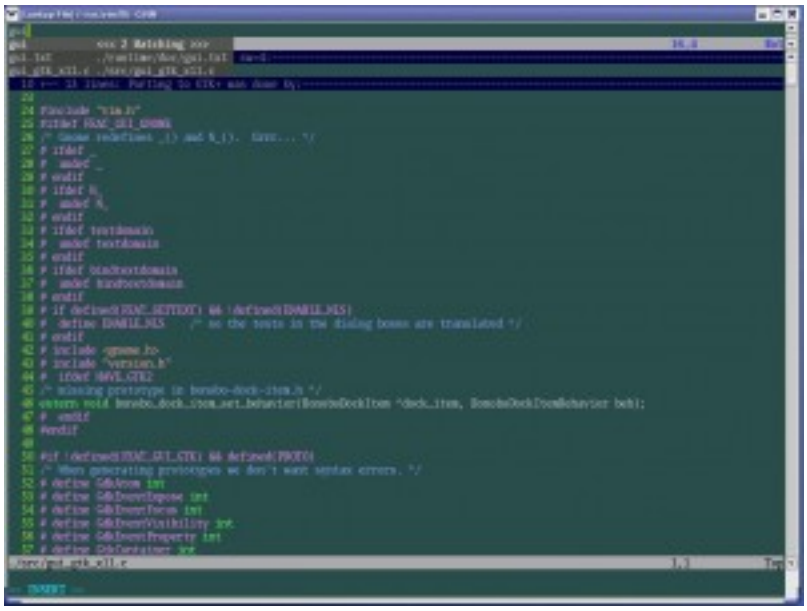
在屏幕最上方的窗口就是lookupfile窗口，在这个窗口中输入"gui.*x11"几个字符，查找到6个匹配文件，使用CTRL-N选中gui_x11.c文件，然后按回车，就会在前一个vim窗口中打开src/gui_x11.c文件，lookupfile窗口也自动关闭了。是不是非常方便？！

[缓冲区查找]

在开发过程中，我经常会同时打开数十甚至上百个文件。即使是使用BufExplorer插件，想在这么多buffer中切换到自己所要的文件，也不是件容易的事。

Lookupfile插件提供了一个按缓冲区名字查找缓冲区的方式，只要输入缓冲区的名字（可以是正则表达式），它就可以把匹配的缓冲区列在下拉列表中，同时还会列出该缓冲区内文件的路径，当你的buffer中有多个同名文件时，这可以帮你迅速找到你想要的文件。

使用":LUBufs"命令开始在缓冲区中查找，输入缓冲区的名字，在你输入的过程中，符合条件的缓冲区就显示在下拉列表中了，选中所需缓冲区后，按回车，就会切换你所选的缓冲区。下图是一个示例：

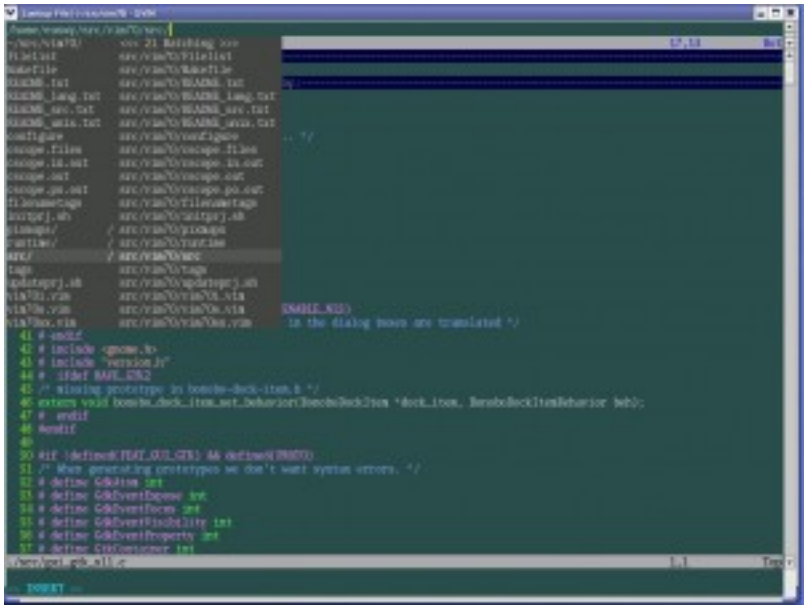


点击查看大图

[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，就可以输入目录，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。

下图是使用LUWalk的一个抓图：



点击查看大图

Lookupfile插件还提供了LUPath和LUArgs两个功能，这两个功能我用的不多，就不在这里介绍了。感兴趣的朋友读一下lookupfile的手册。

[Lookupfile配置]

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。下面是我的vimrc中关于lookupfile的设置，供参考：

```
"""""""""""""""""""""""""""""""
" 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        "不保存上次查找的字符串
let g:LookupFile_PreservePatternHistory = 1     "保存查找历史
let g:LookupFile_AlwaysAcceptFirst = 1          "回车打开第一个匹配项目
let g:LookupFile_AllowNewFiles = 0              "不允许创建不存在的文件
if filereadable("./filenametags")                "设置tag文件的名字
let g:LookupFile_TagExpr = '"./filenametags"'
endif
nmap <silent> <leader>lk <Plug>LookupFile<cr>   "映射LookupFile为,lk
nmap <silent> <leader>ll :LUBufs<cr>            "映射LUBufs为,ll
nmap <silent> <leader>lw :LUWalk<cr>            "映射LUWalk为,lw
```

有了上面的定义，当我输入",lk"时，就会在tag文件中查找指定的文件名；当输入",ll"时，就会在当前已打开的buffer中查找指定名字的buffer；当输入",lw"时，就会在指定目录结构中查找。

另外，我还在项目相关的配置文件vim70sx.vim（参考保存项目相关配置）中加入了lookupfile所使用的tag文件的信息：

```
" lookup file tag file
let g:LookupFile_TagExpr = '"filenametags"'
```

这样，在恢复前次会话时就给lookupfile插件定义了tag文件。

在用lookupfile插件查找文件时，是区分文件名的大小写的，如果想进行忽略大小写的匹配，可以使用vim忽略大小写的正则表达式，即在文件名的前面加上"\c"字符。举个例子，当你输入"\cab.c"时，你可能会得到"ab.c"、"Ab.c"、"AB.c"...

注：如果想加快lookupfile忽略大小写查找的速度，在生成文件名tag文件时，使用混合大小写排序。这在文章使用标签(tag)文件有所提及。

通常情况下我都进行忽略大小写的查找，每次都输入"\c"很麻烦。没关系，lookupfile插件提供了扩展功能，把下面这段代码加入你的vimrc中，就可以每次在查找文件时都忽略大小写查找了：

```
" lookup file with ignore case
function! LookupFile_IgnoreCaseFunc(pattern)
    let _tags = &tags
    try
        let &tags = eval(g:LookupFile_TagExpr)
        let newpattern = '\c' . a:pattern
        let tags = taglist(newpattern)
    catch
        echohl ErrorMsg | echo "Exception: " . v:exception | echohl NONE
        return ""
    finally
        let &tags = _tags
    endtry

    " Show the matches for what is typed so far.
    let files = map(tags, 'v:val["filename"]')
    return files
endfunction
let g:LookupFile_LookupFunc = 'LookupFile_IgnoreCaseFunc'
```

有时在LUBufs时也需要忽略缓冲区名字的大小写，我是通过直接修改lookupfile插件的方法，在LUBufs查找的字符串前都加上"\c"，使之忽略大小写。如果你不想这样，可以每次在缓冲区名字前手动加上"\c"。

本文关于Lookupfile插件就介绍这么多，请阅读手册获取更多的信息。

这是一个非常好的插件，希望能为更多人喜爱！

也欢迎大家多交流使用中的心得和疑问。

[参考文档]

- vim手册
- vim中文手册

<< 返回vim使用进阶: 目录

第 10 章 开启文件类型检测

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

```
:help filetype
:help setfiletype
:help modeline
:help 'modelines'
:help 'shiftwidth'
:help 'expandtab'
:help autocmd
```

打开文件类型检测功能很简单，在你的vimrc中加入下面一句话就可以了：

```
filetype plugin indent on
```

如果你用的是vim自带的示例vimrc，那么你已经打开了文件类型检测功能。或者，你也可以输入":filetype"命令查看你的文件类型检测功能有没有打开。

这条命令究竟做了什么呢？我们在下面详细介绍。

其实，上面这一条命令，可以分为三条命令：

```
filetype on
filetype plugin on
filetype indent on
```

我们逐条介绍这三条命令。

"filetype on"命令打开文件类型检测功能，它相当于文件类型检测功能的开关。在执行"filetype on"命令时，vim实际上执行的是$vimRUNTIME/filetype.vim脚本。这个脚本使用了自动命令(autocmd)来根据文件名来判断文件的类型，如果无法根据文件名来判断出文件类型，它又会调用$vimRUNTIME/scripts.vim来根据文件的内容设置文件类型。有兴趣可以读一下这两个脚本，以获得更深的认识。

在上述步骤完成后，绝大多数已知类型的文件，都能被正确检测出文件类型。如果文件的类型未能被正确的检测出来，就需要手工设置文件类型，这可以通过"set filetype"完成，例如，如果你把main.c改名为main.c.bak1，那么它就无法被正确检测出文件类型。通过下面的Ex命令，就可以把它的文件类型设为c：

```
:set filetype=c
```

或者，你可以在文件中加入一个模式行，来指明这个文件的类型。vim在打开文件时，会在文件首、尾的若干行（行数由'modelines'选项决定，缺省为5行）检测具有vim特殊标记的行，称为模式行。如果检测到，就使用模式行中定义的选项值，来修改该缓冲区的选项。你可以留意一下vim的帮助页，每个文件的最后一行都是这样的模式行。

针对上例，我们可以在main.c.bak1的第一行或最后一行加上下面的内容，要指定这个文件的类型：

```
/* vim: ft=c */
```

这句话使用"/* */"注释起来了，因此不会影响编译。"ft"是"filetype"的缩写，vim中很多命令、选项都有缩写形式，以方便使用。注意"/*"与"vim:"间的空格。在"*/"与"ft=c"间，也需要有至少一个空格，这是模式行格式的要求，更多信息参阅":help modeline"。

检测出文件的类型有什么作用呢？我们知道，不同类型的文件具有不同的格式，vim通过对文件类型的识别，可以为不同类型的文件，设置不同的选项值、定义不同键绑定等。例如，对于c类型的文件，它就和bash脚本有不同的注释格式、不同的缩进格式、不同的关键字等。vim在设置文件类型后，会触发FileType事件，执行FileType相关的自动命令，对不同类型的文件区别对待。

上面提到的"filetype plugin on"，允许vim加载文件类型插件。当这个选项打开时，vim会根据检测到的文件类型，在runtimepath中搜索该类型的所有插件，并执行它们。

"filetype plugin on"命令，实际上是执行$vimRUNTIME/ftplugin.vim脚本，有兴趣可以读一下这个脚本。这个脚本中会设置自动命令，在runtimepath中搜索文件类型插件。

runtimepath的定义在不同的系统上不一样，对UNIX系统来说，这些路径包括：
$HOME/.vim、$vim/vimfiles、$vimRUNTIME、$vim/vimfiles/after、$HOME/.vim/after。

举一个例子，当我们对一个c类型的文件打开"filetype plugin on"时，它会在上述这几个目录的ftplugin子目录中搜索所有名为c.vim、c_*.vim，和c/*.vim的脚本，并执行它们。在搜索时，它按目录在runtimepath中出现的顺序进行搜索。缺省的，它会执行$vimRUNTIME/ftplugin/c.vim，在这个脚本里，会设置c语言的注释格式、智能补全函数等等。

"filetype indent on"允许vim为不同类型的文件定义不同的缩进格式。这条命令也是通过一个脚本来完成加载：$vimRUNTIME/indent.vim。和"filetype plugin on"类似，它也通过设置自动命令，在runtimepath的indent子目录中搜索缩进设置。对c类型的文件来说，它只是打开了cindent选项。

我们了解了文件类型检测的用途及它是如何工作的之后，就可以根据自己的需要，来对特定的文件类型进行特殊设置。

例如，我们在上篇文章中介绍过lookupfile插件，在它打开一个缓冲区时，会把缓冲区的filetype设置为lookupfile，我们可以利用这一点，在这个缓冲区里进行一些特殊的配置。例如，我们创建一个名为lookupfile.vim的文件，其内容为：

```
" close lookupfile window by two <Esc>
nnoremap <buffer> <Esc><Esc> <C-W>q
inoremap <buffer> <Esc><Esc> <Esc><C-W>q
```

它定义了两个局部于缓冲区的键绑定，无论在normal模式还是插入模式，只要连按两次ESC，就关闭当前缓冲区。

把这个文件保存到你的runtimepath所指向任一目录的ftplugin子目录中（一般是放在~/.vim/ftplugin目录中）。你在下次打开lookupfile窗口时，试试连按两次ESC，是不是lookupfile窗口就关闭了？这样设置，非常适合vim中按ESC取消命令的习惯，效率也高。

如果你对vim缺省文件类型插件的设置不太满意，那么可以把这个全局插件拷贝到$HOME/.vim/plugin目录中，然后更改其中的设置。你可以留意一下vim缺省的文件类型插件，它们的头部都有这样的代码：

```
" Only do this when not done yet for this buffer
if exists("b:did_ftplugin")
  finish
endif
```

这类似于C语言头文件中的"#ifndef XXX | #define XXX"的语句，可以防止该插件被执行多次。因此，把这个插件拷贝到$HOME/.vim/plugin中（这个目录在runtimepath中排在最前面），它将先于vim的缺省插件被加载；而它加载后，vim的缺省文件类型插件就不会再被加载了。这就达到了我们修改设置的目的。

不过我们通常不用这种方法。如果这样做，一旦vim的缺省插件做了改变，我们也要更新我们改过的插件才行。我们可以在载入全局插件以后否决一些设置。在Unix 上，我们可以把我们的设置放在~/.vim/after/ftplugin/目录中，这个目录中的脚本会在vim的缺省脚本后执行。这样就可以修改配置，或增加我们自己的定义。

举个例子，在多人一起开发项目时，为了使代码风格尽量保持一致，一般不允许在代码使用TAB符，而以4个空格代之。我们可以编辑一个文件，包含下面的内容：

```
set shiftwidth=4
set expandtab
```

把这个文件保存为~/.vim/after/ftplugin/c.vim。这样，每次在编辑c文件时，它的自动缩进为4个空格；当你在插入模式下使用CTRL-D、CTRL-T缩进时，它也会调整4个空格的缩进；当你按TAB键时，它将会插入8个空格.....如果你想上面的设置对h文件也生效的话，需要把它另存一份：~/.vim/after/ftplugin/cpp.vim，因为h文件的文件类型被设为cpp。

我们知道，vim在设置文件类型时，会触发FileType自动命令，因此，上面的例子可以用下面的自动命令来实现：

```
autocmd FileType c,cpp set shiftwidth=4 | set expandtab
```

把这个命令放在你的vimrc中，可以起到和上例同样的效果。

vim的语法高亮功能，也要用到文件类型，来对不同的关键字进行染色。这我们将在下一篇文章中介绍。

[参考文档]

- vim手册
- vim中文手册

<< 返回vim使用进阶: 目录

第 11 章 乱花渐欲迷人眼 - 语法高亮

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

```
:help syn-enable
:help :colorscheme
:help :highlight
:help highlight-groups
:help 2html.vim
```

看到标题，也许你就知道本文准备讲vim的色彩机制了。

vim并不是只有黑色两色。正相反，它提供了非常灵活的机制允许用户自定义色彩。运行在终端中的vim，由于终端本身的限制，只能使用若干种固定的颜色；但对于gvim来讲，你可以根据你的喜好调出任意的颜色。

首先，把下面的Ex命令加入你的vimrc，打开vim的语法高亮功能：

```
syntax enable
```

这条命令，实际上是执行$vimRUNTIME/syntax/syntax.vim脚本。如果你还没有打开文件类型检测功能，在这个脚本里会把它打开，因为要语法高亮，首先需要知道是什么文件类型。然后它会安装Filetype自动命令，在检测到文件类型时，设置syntax选项。而对syntax选项进行设置，又会触发Syntax自动命令，这条自动命令会在runtimepath的syntax子目录搜寻该类型的语法文件，并使用缺省的配色方案进行染色。

所谓语法文件，就是定义某种类型文件的语法。以C语言为例，它的语法文件定义了什么应该做为关键字来高亮，什么被做为注释来高亮，等等。vim将根据语法文件的定义，把关键字以一种颜色高亮出来，把注释以另一种颜色高亮出来。具备使用什么样的颜色，则由配色方案（colorscheme）来决定，缺省使用default配色方案。

现在，你的世界已经亮起来了：注释、关键字、常数、字符串等等都以不同的颜色显示出来，读程序轻松多了。可是你不太喜欢default配色方案的设置，可不可以换用其它的配色方案呢？

当然没可以！如果你使用的是gvim，在"编辑"菜单中选择"配色方案"，你就可以在多个配色方案中切换了。也可以使用colorscheme命令来改变你的配色方案。例如，我喜欢的GUI配色方案是darkblue，因此我在vimrc中加入这样一句话：

```
colorscheme darkblue
```

这样，进入vim之后，我所用的就是darkblue方案了。

在http://www.vim.org/上还有更多的配色方案，你可以在 http://www.vim.org/scripts/script.php?script_id=625 下载截止到2005年3月所有colorscheme的汇总，你可以在其中找到一个自己喜欢的。安装colorscheme时，只需要把它们拷贝到.vim/color目录下就行了。

有一个名为Color Scheme Explorer的插件，可以帮助你快速浏览你所安装的color scheme，在这里下载：

http://www.vim.org/scripts/script.php?script_id=1298

选择了喜欢的colorscheme后，在vimrc中加入一条colorscheme命令，以后vim就会使用你选定的配色方案了。

如果对配色方案某些颜色不太满意，那么你可以在原来配色方案的基础上，修改其中的一些定义。例如，我把desert.vim拷到.vim/color目录，重命名为darkblue_my.vim。然后做如下更改（只列出改变的内容）：

```
let colors_name = "darkblue_my"
hi Normal guifg=#c0c0c0 guibg=#294d4a ctermfg=gray ctermbg=black
......
"Omni menu colors
hi Pmenu guibg=#444444
hi PmenuSel ctermfg=7 ctermbg=4 guibg=#555555 guifg=#ffffff
" Matched brackets
hi MatchParen ctermfg=7 ctermbg=4
```

首先改变colors_name，vim在某此情况会根据这个名字重新载入color scheme。

接下来我重新设置了GUI的背景色，在前面的抓图中大家看到过这个颜色。

接下来的Pmenu和PmenuSel用来设置vim下拉菜单的颜色，我们在使用lookupfile插件中看到过下拉菜单。

vim 7中，当光标移到括号上时，vim会高亮与之匹配的括号，所使用的颜色就是MatchParen，我在这里也更改这个颜色。

Pmenu、PmenuSel，以及MatchParen，都是vim定义的缺省高亮组的名字，你可以用":help highlight-groups"命令查看有这些高亮组及其含义。

如果你打算在终端及GUI界面中使用不同的colorscheme，可以这样设：

```
" color scheme
if has("gui_running")
  colorscheme darkblue_my
else
  colorscheme desert_my
endif " has
```

这里的darkblue_my和desert_my都是我自己改过的colorscheme。

vim还提供了一个脚本，可以把你的文件按当前的颜色定义转化成HTML/XML文件，试试":TOhtml"命令吧，更多信息请":help 2html.vim"。

记得Source Insight中有一个功能，按SHIFT+F8可以把光标下的词高亮出来，在看代码时非常有用。vim下也有一个插件可以完成此功能，而且比Source Insight的这个功能强大多了。

这个插件由Yuheng Xie所写，对这个插件有什么疑问，可以水木社区的vim版找到他（http://www.newsmth.net/bbsdoc.php?board=vim）。在这里下载此插件。

把此插件直接拷贝到你的.vim/plugin目录就行了。

我在vimrc中这样设置：

```
""""""""""""""""""""""""""""""
```

```
" mark setting
""""""""""""""""""""""""""""""
nmap <silent> <leader>hl <Plug>MarkSet
vmap <silent> <leader>hl <Plug>MarkSet
nmap <silent> <leader>hh <Plug>MarkClear
vmap <silent> <leader>hh <Plug>MarkClear
nmap <silent> <leader>hr <Plug>MarkRegex
vmap <silent> <leader>hr <Plug>MarkRegex
```

这样，当我输入",hl"时，就会把光标下的单词高亮，在此单词上按",hh"会清除该单词的高亮。如果在高亮单词外输入",hh"，会清除所有的高亮。

你也可以使用virsual模式选中一段文本，然后按",hl"，会高亮你所选中的文本；或者你可以用",hr"来输入一个正则表达式，这会高亮所有符合这个正则表达式的文本。

你可以在高亮文本上使用",#"或",*"来上下搜索高亮文本。在使用了",#"或",*"后，就可以直接输入"#"或"*"来继续查找该高亮文本，直到你又用"#"或"*"查找了其它文本。

如果你在启动vim后重新执行了colorscheme命令，或者载入了会话文件，那么mark插件的颜色就会被清掉，解决的办法是重新source一下mark插件。或者像我一样，把mark插件定义的highlight组加入到你自己的colorscheme文件中。例如，把下面的语句加到desert_my.vim及darkblue_my.vim中：

```
" For mark plugin
hi MarkWord1  ctermbg=Cyan     ctermfg=Black  guibg=#8CCBEA    guifg=Black
hi MarkWord2  ctermbg=Green    ctermfg=Black  guibg=#A4E57E    guifg=Black
hi MarkWord3  ctermbg=Yellow   ctermfg=Black  guibg=#FFDB72    guifg=Black
hi MarkWord4  ctermbg=Red      ctermfg=Black  guibg=#FF7272    guifg=Black
hi MarkWord5  ctermbg=Magenta  ctermfg=Black  guibg=#FFB3FF    guifg=Black
hi MarkWord6  ctermbg=Blue     ctermfg=Black  guibg=#9999FF    guifg=Black
```

不知道为什么，我的vim 7.0在切换到其它缓冲区然后再切换回来时，原来被标记的文本会失去高亮。而作者说他并不存在此问题。如果你存在类似的问题，可以打上我所加的补丁：

```
--- easwy/mark.vim 2006-12-01 13:02:18.000000000 +0800
+++ plugin/mark.vim 2007-03-23 10:22:02.000000000 +0800
@@ -440,6 +440,43 @@
  endif
 endfunction

+" easwy add
+" return the mark string under the cursor. multi-lines marks not supported
+function! <SID>RedoMarkWord()
+ " define variables if they don't exist
+ call s:InitMarkVariables()
+
+ let i = 1
+  while i <= g:mwCycleMax
+    if b:mwWord{i} != ""
+      " quote regexp with / etc. e.g. pattern => /pattern/
+      let quote = "/?~!@#$%^&*+-=,.:"
+      let j = 0
```

```
+      while j < strlen(quote)
+        if stridx(b:mwWord{i}, quote[j]) < 0
+          let quoted_regexp = quote[j] . b:mwWord{i} . quote[j]
+          break
+        endif
+        let j = j + 1
+      endwhile
+      if j >= strlen(quote)
+        return -1
+      endif
+
+      " highlight the word
+      exe "syntax clear MarkWord" . i
+      exe "syntax match MarkWord" . i . " " . quoted_regexp . " containedin=ALL"
+    endif
+    let i = i + 1
+  endwhile
+endfunction
+
+augroup markword
+  autocmd!
+  autocmd BufWinEnter * call <SID>RedoMarkWord()
+augroup END
+" easwy end
+
 " Restore previous 'cpo' value
 let &cpo = s:save_cpo
```

用法：

1. 保存该patch到某一目录，例如：/tmp/mark.vim.patch

2. cd到你的.vim目录：cd ~/.vim

3. 运行命令：cat /tmp/mark.vim.patch | patch -p0

[参考文档]

- vim手册

- vim中文手册

<< 返回vim使用进阶: 目录

原创文章，转载请注明：转载自Easwy的博客 [http://easwy.com/blog/]

本文链接地址: http://easwy.com/blog/archives/advanced-vim-skills-syntax-on-colorscheme/

第 12 章 程序员的利器 - cscope

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

```
:help cscope
```

在前面的文章中介绍了利用tag文件，跳转到标签定义的地方。但如果想查找函数在哪里被调用，或者标签在哪些地方出现过，ctags就无能为力了，这时需要使用更为强大的cscope。

Cscope具有纯正的Unix血统，它最早是由贝尔实验室为PDP-11计算机开发的，后来成为商用的AT&T Unix发行版的组成部分。直到2000年4月，这个工具才由SCO公司以BSD license开源发行。

Cscope的主页在http://cscope.sourceforge.net/，如果你的计算机上没有cscope，你可以在此处下载它，在写本文时，它的最新版本是15.6。安装它非常简单，你只需要在cscope的源代码目录中执行下面三条命令：

```
./configure
make
make install
```

在windows上也可以使用cscope，在cscope的主页上可以下载到由DJGPP编译器编译的cscope for windows，不过这个版本不能和vi一起工作。或者你可以下载cygwin工具包(http://www.cygwin.com/)，这个工具包中也包含了cscope。

在http://iamphet.nm.ru/cscope/有Sergey Khorev预编译的一个Win32版本的cscope，这个版本的cscope可以很好的与windows版本的vim搭配使用。

cscope的用法很简单，首先需要为你的代码生成一个cscope数据库。在你的项目根目录运行下面的命令：

```
cscope -Rbq
```

这些选项的含义见后面。这个命令会生成三个文件：cscope.out，cscope.in.out，cscope.po.out。其中cscope.out是基本的符号索引，后两个文件是使用"-q"选项生成的，可以加快cscope的索引速度。

在缺省情况下，cscope在生成数据库后就会进入它自己的查询界面，我们一般不用这个界面，所以使用了"-b"选项。如果你已经进入了这个界面，按CTRL-D退出。

Cscope在生成数据库中，在你的项目目录中未找到的头文件，会自动到/usr/include目录中查找。如果你想阻止它这样做，使用"-k"选项。

Cscope缺省只解析C文件(.c和.h)、lex文件(.l)和yacc文件(.y)，虽然它也可以支持C++以及Java，但它在扫描目录时会跳过C++及Java后缀的文件。如果你希望cscope解析C++或Java文件，需要把这些文件的名字和路径保存在一个名为cscope.files的文件。当cscope发现在当前目录中存在cscope.files时，就会为cscope.files中列出的所有文件生成索引数据库。通常我们使用find来生成cscope.files文件，仍以vim 7.0的源代码为例：

```
cd ~/src/vim70
```

```
find .  -type f > cscope.files
cscope -bq
```

这条命令把~src/vim70目录下的所有普通文件都加入了cscope.files，这样，cscope会解析该目录下的每一个文件。上面的cscope命令并没有使用"-R"参数递归查找子目录，因为在cscope.files中已经包含了子目录中的文件。

注意：find命令输出的文件以相对路径表示，所以cscope.out的索引也相对于当前路径。如果你要在其它路径中使用当前的cscope.out，需要使用下面介绍的-P选项。

Cscope只在第一次解析时扫描全部文件，以后再调用cscope，它只扫描那些改动过的文件，这大大提高了cscope生成索引的速度。

下表中列出了cscope的常用选项：

- -R: 在生成索引文件时，搜索子目录树中的代码
- -b: 只生成索引文件，不进入cscope的界面
- -q: 生成cscope.in.out和cscope.po.out文件，加快cscope的索引速度
- -k: 在生成索引文件时，不搜索/usr/include目录
- -i: 如果保存文件列表的文件名不是cscope.files时，需要加此选项告诉cscope到哪儿去找源文件列表。可以使用"-"，表示由标准输入获得文件列表。
- -Idir: 在-I选项指出的目录中查找头文件
- -u: 扫描所有文件，重新生成交叉索引文件
- -C: 在搜索时忽略大小写
- -Ppath: 在以相对路径表示的文件前加上的path，这样，你不用切换到你数据库文件所在的目录也可以使用它了。

要在vim中使用cscope的功能，需要在编译vim时选择"+cscope"。vim的cscope接口先会调用cscope的命令行接口，然后分析其输出结果找到匹配处显示给用户。

在vim中使用cscope非常简单，首先调用"cscope add"命令添加一个cscope数据库，然后就可以调用"cscope find"命令进行查找了。vim支持8种cscope的查询功能，如下：

- s: 查找C语言符号，即查找函数名、宏、枚举值等出现的地方
- g: 查找函数、宏、枚举等定义的位置，类似ctags所提供的功能
- d: 查找本函数调用的函数
- c: 查找调用本函数的函数
- t: 查找指定的字符串
- e: 查找egrep模式，相当于egrep功能，但查找速度快多了
- f: 查找并打开文件，类似vim的find功能
- i: 查找包含本文件的文件

例如，我们想在vim 7.0的源代码中查找调用do_cscope()函数的函数，我们可以输入：":cs find c do_cscope"，回车后发现没有找到匹配的功能，可能并没有函数调用do_cscope()。我们

再输入":cs find s do_cscope"，查找这个C符号出现的位置，现在vim列出了这个符号出现的所有位置。

我们还可以进行字符串查找，它会双引号或单引号括起来的内容中查找。还可以输入一个正则表达式，这类似于egrep程序的功能，但它是在交叉索引数据库中查找，速度要快得多。

vim提供了一些选项可以调整它的cscope功能：

- cscopecscopeprg选项用于设置cscope程序的位置。
- cscopecscopequickfix设定是否使用quickfix窗口来显示cscope的结果，详情请":help cscopequickfix"；
- 如果你想vim同时搜索tag文件以及cscope数据库，设置cscopecscopetag选项；
- cscopecscopetagorder选项决定是先查找tag文件还是先查找cscope数据库。设置为0则先查找cscope数据库，设置为1先查找tag文件。我通常设置为1，因为在tag文件中查找到的结果，会把最佳匹配列在第一位。

vim的手册中给出了使用cscope的建议方法，使用命令":help cscope-suggestions"查看。

下面是我的vimrc中关于cscope接口的设置：

```
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
" cscope setting
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
if has("cscope")
  set csprg=/usr/bin/cscope
  set csto=1
  set cst
  set nocsverb
  " add any database in current directory
  if filereadable("cscope.out")
      cs add cscope.out
  endif
  set csverb
endif

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>
```

下面的两个链接是cscope主页提供的cscope使用方法，也可以作为参考：

vim/cscope指导：http://cscope.sourceforge.net/cscope_vim_tutorial.html

在大项目中使用cscope：http://cscope.sourceforge.net/large_projects.html

在vim的网站上有很多与cscope相关的插件，有兴趣可以去看一下。

我以前写的Vim + Cscope/Ctags。

[参考文档]

- vim帮助文件
- vim中文手册
- http://cscope.sourceforge.net/
- http://iamphet.nm.ru/cscope/
- cscope手册

<< 返回vim使用进阶: 目录

第 13 章 剑不离手 - quickfix

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

```
:help quickfix
:help :make
:help 'makeprg'
:help 'errorformat'
:help 'switchbuf'
:help location-list
:help grep
:help :vimgrep
:help :grep
:help starstar-wildcard
```

以前读武侠小说，看到武林高手们都是从来剑不离手的。使用vim写程序，你也可以做到这一点，:-)

vim由一个程序员开发，而且为更多的程序员所使用，所以在vim中加强了对软件开发的支持，quickfix模式的引入就是一个例子。所谓quickfix模式，它和Normal模式、Insert模式没什么关系，它只是一种加快你开发速度的工作方式。

Quickfix模式的主要思想是保存一个位置列表，然后提供一系列命令，实现在这个位置列表中跳转。

位置列表的产生可以从编译器的编译输出信息中获得，也可以由grep命令的输出信息中获得，我们上篇文章所介绍的cscope命令，也可以产生位置列表信息(:help 'cscopequickfix')。

[编译]

通常，我们在开发过程中，经常要写代码，编译，修改编译错误，这个过程会数十遍上百遍的重复。如果你是根据编译器输出的错误信息，打开出错的文件，找到出错的行，然后再开始修改，那效率未免太低下了。

利用vim的quickfix模式，可以大大加快这一过程，你可以在vim启动编译，然后vim会根据编译器输出的错误信息，自动跳到第一个出错的地方，让你进行修改；修改完后，使用一个快捷键，跳到下一个错误处，再进行修改，方便的很。

为了做到这一点，你首先要定义编译时所使用的程序，对大多数使用Makefile的项目来说，vim的缺省设置"make"已经可以满足要求了。如果你的项目需要用一个特殊的程序进行编译，就需要修改'makeprg'选项的值。

大家在学编程时大概都读过"hello world"程序，我们就以这个简单的例子为例，讲一下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：

```
/* hello world demo */
#include <stdio.h"
int main(int argc, char **argv)
```

```
{
    int i;
    print("hello world\n");
    return 0;
}
```

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

```
:set makeprg=gcc\ -Wall\ -ohello\ hello.c
```

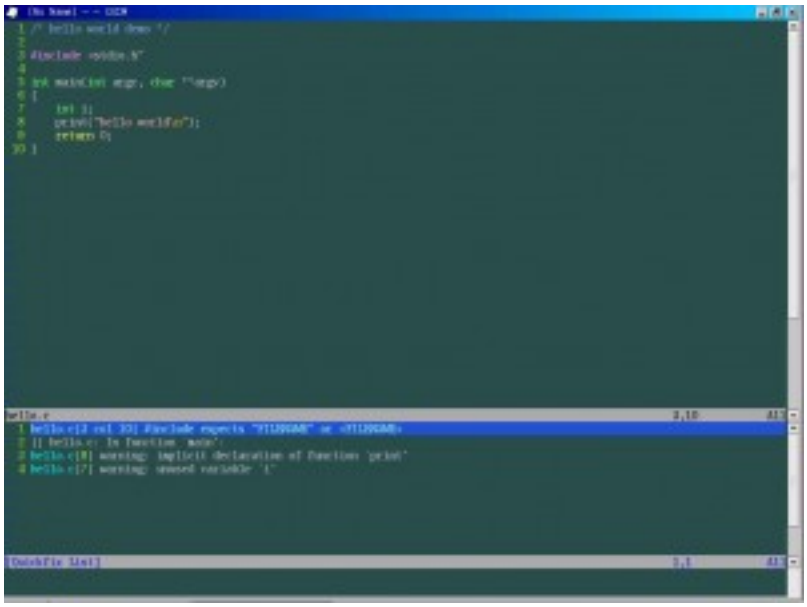
上面的命令会把hello.c编译为名hello的可执行文件，并打开了所有的Warnning。如果编译命令中有空格，需要使用'\'对空格进行转义，上面的例子就使用了'\'转义空格。

我们设置好'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。

对于我们上面的程序来说，光标会停在第三行，也就是第一个出错的位置，vim同时会提示出错信息。如果你没看清出错信息，可以输入":cc"命令，vim会更次显示此信息，或者干脆使用":cw"命令，打开一个quickfix窗口，把所有的出错信息显示出来，见下图：



点击查看大图

现在我们知道错在哪儿了，修正一下，然后使用":cn"命令（或者在Quickfix List对应行上输入回车）跳到下一个出错的地方，以此类推，直到修正全部错误。

好了，千辛万苦，我们的hello world终于工作了。乍一看这个例子，似乎Quickfix并没有提高什么效率，但如果你的错误出现在多个不同目录的不同文件里，它可以帮你省很多时间，使你可以集中精力在修正bug上。

vim可以同时记住最新的10个错误列表，也就是说你最近10次使用":make"命令编译所遇到的错误都保存着，可以使用":colder"和":cnewer"命令，回到旧的错误列表，或者到更新的错误列表。

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 )
```

更多的命令，以及这些命令更详细的解释，请参见手册。

对于经常用到的命令，最好提供更方便的使用方法，在我的vimrc中的定义：

```
autocmd FileType c,cpp  map <buffer> <leader><space> :w<cr>:make<cr>
nmap <leader>cn :cn<cr>
nmap <leader>cp :cp<cr>
nmap <leader>cw :cw 10<cr>
```

现在使用",<space>"(先按,再按空格)就可以编译，使用",cp"和",cn"跳到上一个和下一个错误，使用",cw"来打开一个quickfix窗口。这下顺手多了！

如果你希望跳转到出错的文件时，使用一个分隔的窗口打开，请参阅'switchbuf'选项的值。

在vim7中，每个窗口都可以拥有自己的位置列表，这样，你就能够同时打开多个位置列表了，而quickfix列表在整个vim中只有一个。你可以使用位置列表来显示编译错误信息，具体命令参阅手册：":help location-list"以及":help :lmake"。

[GREP]

我们在程序员的利器 - cscope中讲过，cscope可以做为一个快速的grep程序使用，对于我们的软件项目，用cscope生成一个数据库，可以大大加快查找字符串的速度。但cscope需要事先生成一个数据库，对一些简单的查找，不需要专门为之生成数据库，这时候可以使用grep。

Grep的名字来源于"g/re/p"，"re"是正则表达式(regex)的意思，"p"是打印，也就是把匹配正则表达式的行打印出来。

vim既可以使用外部的grep程序，也可以使用内部集成的grep功能。

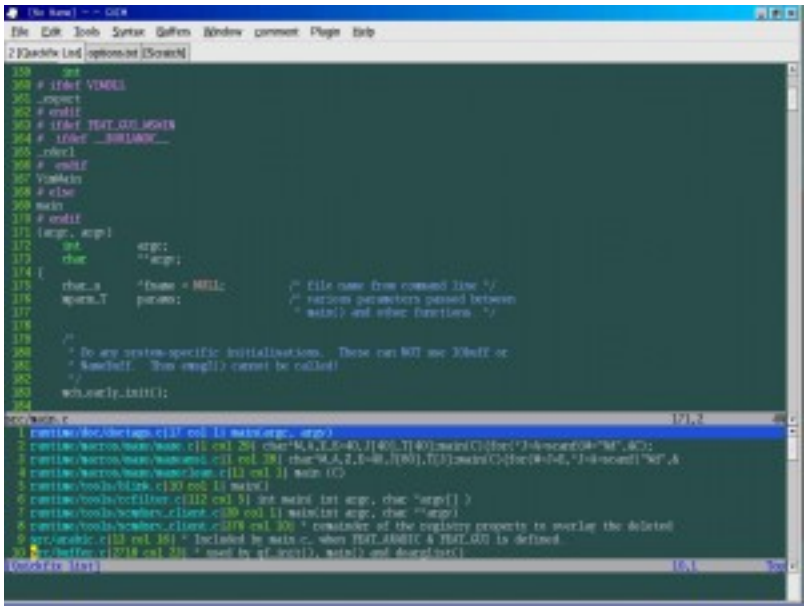
使用集成的grep命令非常简单，通常使用格式为：

```
:vimgrep /main/gj **/*.c
```

在上面的例子里，我们使用vim内部集成的grep功能，在当前目录及其子目录树的所有c文件中查找main字符串，如果一行中main出现了多次，每个匹配都计入；在查找到后，不立即跳转到第一个匹配的地方。

使用内部集成的grep功能速度要比外部grep慢一些，因为它会打开每个文件，对其进行检查，然后关闭；但集成的grep支持vim增强的正则表达式，可以利用它进行更为复杂的查找。它也支持vim扩展的文件通配符表示方式，见":help starstar-wildcard"。

vimgrep查找到的结果，也会放在quickfix列表中。下图是在vim 7.0的源代码目录中执行上面的命令生成的quickfix列表：



点击查看大图

我们可以使用上面介绍的quickfix模式的命令，来查看这些匹配。

你也可以用外部的grep程序来查找，如果你的系统中所用的不是标准的grep程序，那么就需要修改'grepprg'选项，详情请参阅手册。

使用外部grep的语法与grep程序相同，请参阅grep的手册。

无论使用内部的vimgrep，还是使用外部的grep，vim都允许你将查找到的结果放在与窗口相关联的位置列表，要了解详细信息，":help :lvimgrep"及":help :lgrep"。

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>
```

[参考文档]

- vim手册
- vim中文手册

<< 返回vim使用进阶: 目录

第 14 章 智能补全

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

```
:help ins-completion
:help compl-omni
:help 'omnifunc'
:help i_CTRL-X_CTRL-O
:help ins-completion-menu
:help popupmenu-keys
:help 'completeopt'
:help compl-omni-filetypes
:help omnicppcomplete.txt
```

使用过Source Insight的人一定对它的自动补全功能印象深刻，在很多的集成开发环境中，也都支持自动补全。vim做为一个出色的编辑器，这样的功能当然少不了。而且，作为一个通用的编辑器，vim实现的补全功能并不仅仅限于对程序的补全，它可以对文件名补全、根据字典进行补全、根据本缓冲区或其它缓冲区类似的内容进行补全、根据文件语法补全等等，它甚至允许用户自己编写函数来实现定制的补全。

作为vim进阶系列文章中的一篇，本文以介绍vim对程序的补全为主，也顺带介绍一下其它的补全方式。本文将分为两篇，第一篇主要介绍vim的OMNI补全，下一篇简要介绍其它的补全方式，以及SuperTab插件。

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。

vim在对不同类型的文件进行补全时，会根据文件类型，为其设置不同的补全函数。也就是说，要实现全能补全功能，需要打开文件类型检测。把下面的命令加到你的vimrc中：

```
filetype plugin indent on
```

你可以查看'omnifunc'选项，来知道当前的补全函数是什么。

对C及C++代码的全能补全需要使用Exuberant ctags生成的标签文件，我们在前面的文章中介绍过如何使用Exuberant ctags程序来生成标签文件。不过，如果你的Exuberant ctags版本为5.5.4，那么需要为其打上增加"typename:"字段补丁，才能支持C的全能补全。补丁在这里下载：

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

可以在这里找到MS-Windows上已经编译好的可执行版本：

```
http://georgevreilly.com/vim/ctags.html
```

不过我建议使用最新5.6版本Exuberant Ctags。在下面的网站可以下载：

http://ctags.sourceforge.net/

你可以直接下载已经编译好的rpm版本，或者下载源代码。如果是后者，使用以下命令对源代码进行编译：

```
tar zvxf ctags-5.6.tar.gz
cd ctags-5.6
./configure
make
make install
```

如果你没有系统目录的写权限，你可能要把Exuberant Ctags安装到自己的主目录，只需要把上面的"./configure"命令改为"./configure -prefix=/home/xxx"就可以了。

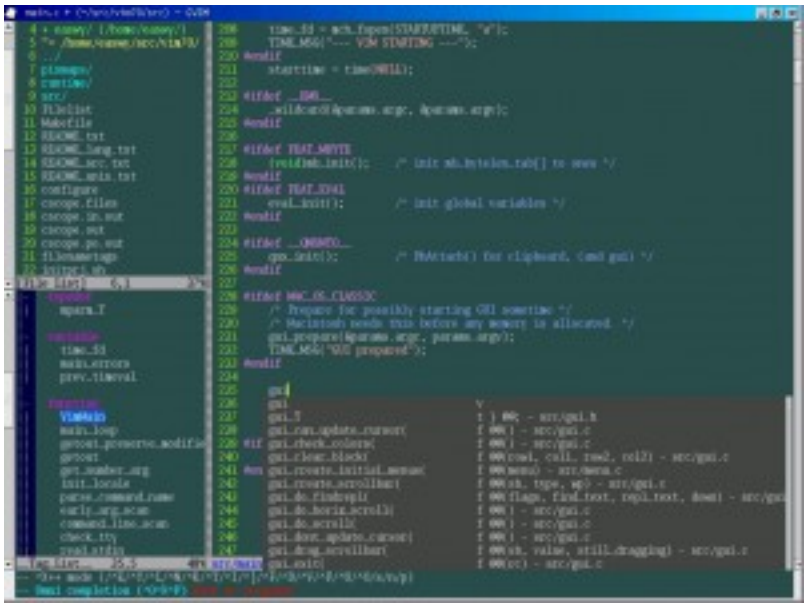
Ctags升级后，使用"ctags -R"更新一下标签文件，现在再进入vim就可以在C程序中全能补全了。我们依旧以vim 7.0的源代码为例。

例如，我们在VimMain()函数中，输入"gui"三个字符，然后按下"CTRL-X CTRL-O"，在vim的状态行会显示"Omni Completeion"，表明现在进行的是全能补全，同时会弹出一个下拉菜单，显示所有匹配的标签。你可以使用来"CTRL-P"和"CTRL-N"上下选择，在选择的同时，所选中的项就被放在光标位置，不需要再按回车来把它放在光标位置（像Source Insight那样）。

如果更习惯于使用Source Insight这种方式，你可以使用上、下光标键来选择项目，然后按回车把选中的项目放到光标位置。不过这样一来，你的手指就会离开主编辑区，并且需要多输入一个回车键。

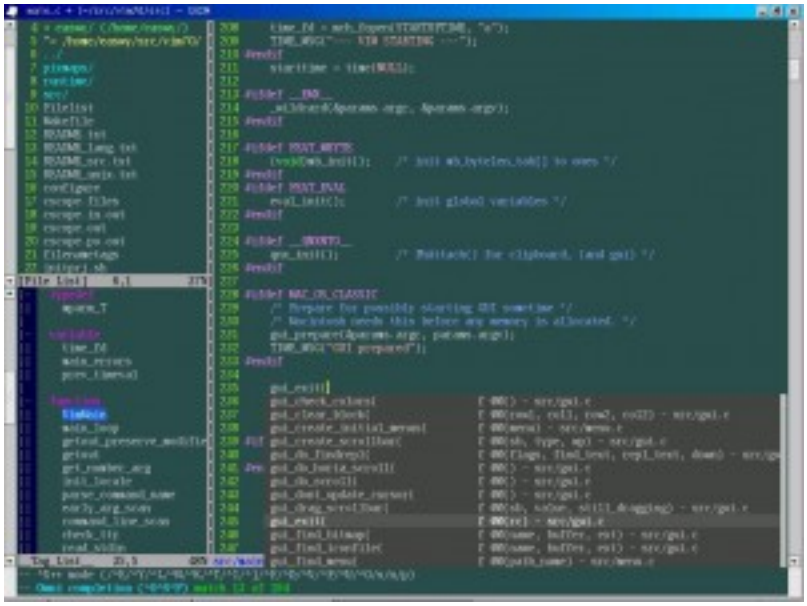
本文结尾处提供了一个键绑定，允许在使"CTRL-P"和"CTRL-N"时，输入回车表示补全结束，而不是插入回车。

如果补全处于激活状态，可以用"CTRL-E"停止补全并回到原来录入的文字。用"CTRL-Y"可以停止补全，并接受当前所选的项目。



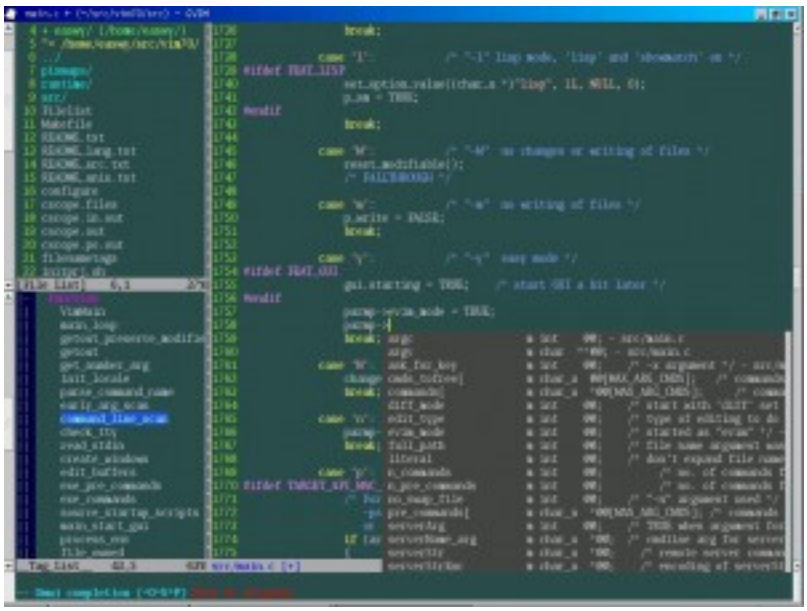
点击查看大图

下图是使用"CTRL-N"选择的抓图。该图中，我选择了"gui_exit("函数，接下来可以直接输入这个函数的参数，这会结束当前补全，并插入我所输入的参数。



点击查看大图

下图是对结构的成员进行补全的抓图：



点击查看大图

缺省的，vim会使用下拉菜单和一个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。打开预览窗口会导致下拉菜单抖动，因此我一般都去掉预览窗口的显示，这需要改变'completeopt'的值，我的设置如下：

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

上面的设置表明，只在下拉菜单中显示匹配项目，并且会自动插入所有匹配项目的相同文本。

如果要支持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
```

下载后，把它解压到你的.vim目录(在windows下是vimfiles目录)，它会安装以下文件：

```
after\ftplugin\cpp.vim
autoload\omni\common\debug.vim
\utils.vim
autoload\omni\cpp\complete.vim
\includes.vim
\items.vim
\maycomplete.vim
\namespaces.vim
\settings.vim
\tokenizer.vim
\utils.vim
doc\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，使用下面的命令，为C++文件生成标签文件，假定你的文件在src目录树下：

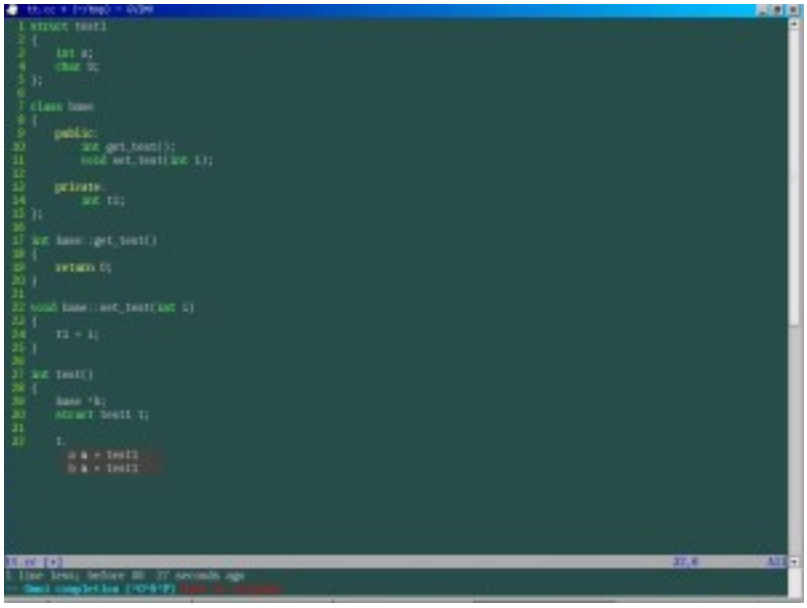
```
ctags -R --c++-kinds=+p --fields=+iaS --extra=+q src
```

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：

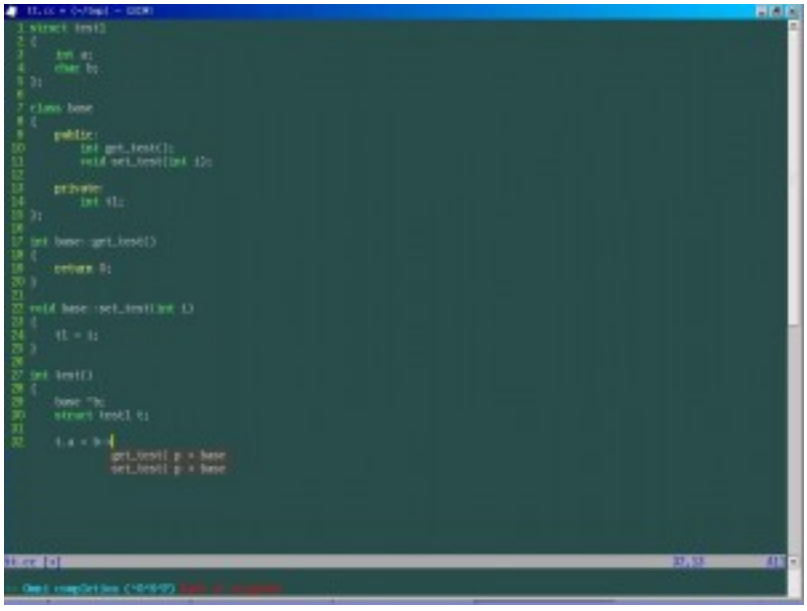
```
--c++-kinds=+p  : 为C++文件增加函数原型的标签
--fields=+iaS   : 在标签文件中加入继承信息(i)、类成员的访问控制信息(a)、以及函数的指纹(S)
--extra=+q      : 为标签增加类修饰符。注意，如果没有此选项，将不能对类成员补全
```

现在，进入vim，设置好tag选项（我在前面的文章中介绍过）。好极了，vim能够对C++自动补全了！

我写了一个简单的例子，来演示C++的自动补全功能，如下图所示，在输入"t."后，OmniCppComplete插件会自动弹出struct test1的成员供选择，而在输入"b->"后，又会自动弹出class base的成员供选择，非常方便，连"CTRL-X CTRL-O"都不必输入。OmniCppComplete插件的缺省设置比较符合我的习惯，因此不须对其设置进行调整，如果你需要调整，参阅OmniCppComplete的帮助页。



点击查看大图



点击查看大图

下表是我的vimrc中设置的键绑定，使用pumvisible()来判断下拉菜单是否显示，如果下拉菜单显示了，键映射为了一个值，如果未显示，又会映射为另一个值。

```
" mapping
inoremap <expr> <CR>       pumvisible()?"\<C-Y>":"\<CR>"
inoremap <expr> <C-J>      pumvisible()?"\<PageDown>\<C-N>\<C-P>":"\<C-X><C-O>"
inoremap <expr> <C-K>      pumvisible()?"\<PageUp>\<C-P>\<C-N>":"\<C-K>"
inoremap <expr> <C-U>      pumvisible()?"\<C-E>":"\<C-U>"
```

上面的映射都是在插入模式下的映射，解释如下：

- 如果下拉菜单弹出，回车映射为接受当前所选项目，否则，仍映射为回车；

- 如果下拉菜单弹出，CTRL-J映射为在下拉菜单中向下翻页。否则映射为CTRL-X CTRL-O；
- 如果下拉菜单弹出，CTRL-K映射为在下拉菜单中向上翻页，否则仍映射为CTRL-K；
- 如果下拉菜单弹出，CTRL-U映射为CTRL-E，即停止补全，否则，仍映射为CTRL-U；

在下一篇文章中，将继续介绍vim提供的其它补全方式。

[参考文档]

- vim手册
- vim中文手册
- http://www.vim.org/tips/tip.php?tip_id=1228
- http://www.vim.org/tips/tip.php?tip_id=1386

<< 返回vim使用进阶: 目录

第 15 章 自动补全

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

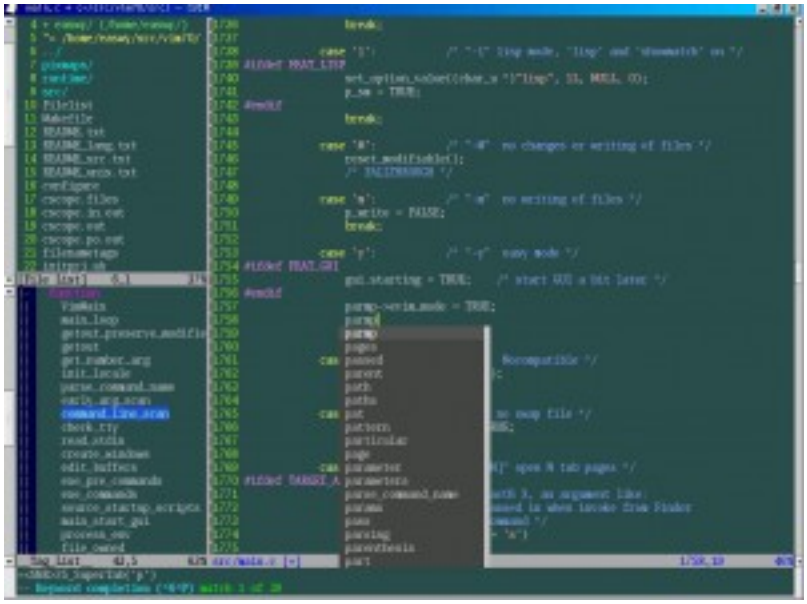
```
:help compl-generic
:help 'complete'
:help ins-completion
```

上篇文章介绍了vim的智能补全(omni补全)，本篇主要介绍vim提供的其它补全方式。

除智能补全外，最常用的补全方式应该是CTRL-N和CTRL-P补全了。它们会在当前缓冲区、其它缓冲区，以及当前文件所包含的头文件中查找以光标前关键字开始的单词。智能补全不能对局部变量进行补全，而CTRL-N和CTRL-P补全则可以很好的胜任。

下图是采用CTRL-P补全的一个例子，输出字符"pa"，然后按CTRL-P，vim会在下拉菜单中列出所有的匹配功能供选择，此时再按一下CTRL-P，就选中了第一个项目，也就是我想输入的"parmp"。我们第一次输入CTRL-P的是进行补全，第二次输入的CTRL-P是在下拉菜单中向上选择，二者的含义是不同的。

我们知道，CTRL-P一般的含义是向上，因此CTRL-P补全是向上查找以进行补全，而CTRL-N是向下查找以进行补全，在不同场合使用不同的快捷键可以加速补全的速度。



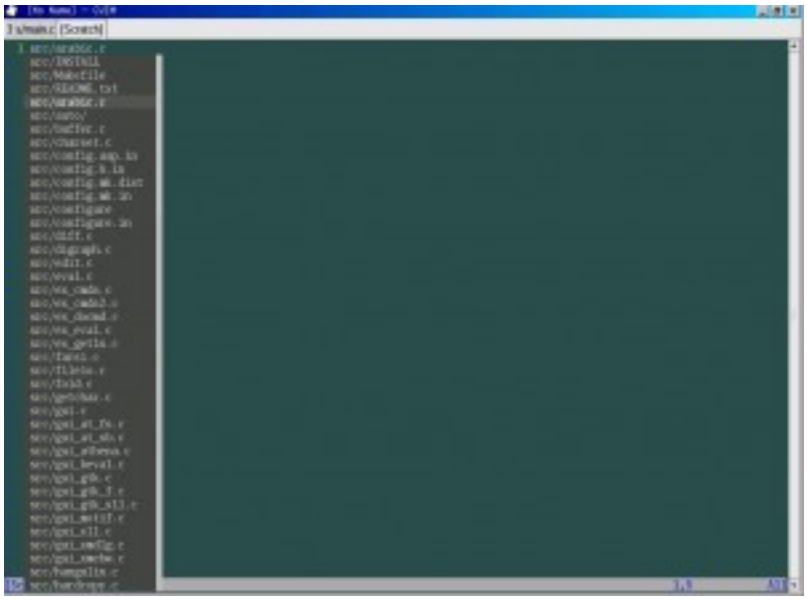
点击查看大图

使用CTRL-N和CTRL-P补全时，由'complete'选项控制vim从哪些地方查找补全的内容。例如，对于比较大的软件项目，文件包含关系复杂，如果CTRL-N和CTRL-P补全时查找所包含的头文件，耗时会比较久。此时，可以在'complete'选项中去掉'i'标记，这样CTRL-N和CTRL-P补全就不在头文件中查找了，速度会快很多；当然，弊端就是你无法对头文件中出现的某些内容进行补全了。'complete'选项中其它标记的含义，请阅读手册页。

vim中其它的补全方式包括：

整行补全	CTRL-X CTRL-L
根据当前文件里关键字补全	CTRL-X CTRL-N
根据字典补全	CTRL-X CTRL-K
根据同义词字典补全	CTRL-X CTRL-T
根据头文件内关键字补全	CTRL-X CTRL-I
根据标签补全	CTRL-X CTRL-]
补全文件名	CTRL-X CTRL-F
补全宏定义	CTRL-X CTRL-D
补全vim命令	CTRL-X CTRL-V
用户自定义补全方式	CTRL-X CTRL-U
拼写建议	CTRL-X CTRL-S

例如，当我们按下"CTRL-X CTRL-F"时，vim就会弹出下拉菜单，显示出当前目录下的可选目录和文件，如下图所示。这样，在输入文件名时方便多了。



点击查看大图

灵活的运用这些补全方式，甚至自定义自己的补全方式，可以使你的工作更加高效。

可以在vimrc中定义下面的键绑定，以减少按键次数：

```
inoremap <C-]>          <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

下载后，把它放到.vim/plugin目录就可以了。

可以对下面两个选项进行配置，以调整SuperTab的缺省行为：

- g:SuperTabRetainCompletionType的值缺省为1，意为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，意味着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，意味着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。

你可以在vimrc中设置这两个变量，例如：

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

现在你可以使用TAB来进行补全了，就像在shell中那样，方便了很多！

[参考文档]

- vim手册

- vim中文手册

<< 返回vim使用进阶: 目录

第 16 章 指随意动，移动如飞（一）

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

```
:help usr_03.txt
:help motion.txt
:help usr_29.txt
:help scroll.txt
```

vim提供的移动方式多如牛毛，但我们并不需要掌握全部这些命令，只需要掌握最适合自己的那些命令。因为我们最终的目的，并不是成为一个vim高手，而是更高效的编辑文本。

我们下面介绍的命令，如果没有特别说明，都是在Normal模式下使用的命令。

这些命令的帮助入口，就是":help 命令名"。例如，对于"j"命令，查看它的帮助，使用":help j"。

[上下左右]

让我们从最简单的、也是使用频率最高的h, j, k, l开始。

h, j, k, l的移动方式，已经成为vim的标志之一，并且也为更多的软件所接受。如果你仍在用上下左右光标来移动的话，说明你内心并没有真正接受vim的哲学，如果真的打算把vim做为你的编辑工具，就从使用h, j, k, l开始吧！

h, j, k, l分别代表向左、下、上、右移动。如同许多vim命令一样，可以在这些键前加一个数字，表示移动的倍数。例如，"10j"表示向下移动10行；"10l"表示向右移动10列。

缺省情况下，h和l命令不会把光标移出当前行。如果已经到了行首，无论按多少次h键，光标始终停留在行首，l命令也类似。如果希望h和l命令可以移出当前行，更改'whichwrap'选项的设置(:help 'whichwrap')。

vim的作者在安排按键功能时别具匠心，在其它的键绑定中，也能看到h, j, k, l所代表的含义。

例如，使光标在多个窗口间上下左右移动的命令，就是CTRL-W h/j/k/l (:help CTRL-W_h, ...)；

再如，上下左右移动窗口位置的命令，是CTRL-W H/J/K/L (:help CTRL-W_H, ...)。注意，这里的H, J, K, L是大写的。

[翻页]

在vim中翻页，同样可以使用PageUp和PageDown，不过，像使用上下左右光标一样，你的手指会移出主键盘区。因此，我们通常使用CTRL-B和CTRL-F来进行翻页，它们的功能等同于PageUp和PageDown。CTRL-B和CTRL-F前也可以加上数字，来表示向上或向下翻多少页。

vim中还可以向上或向下翻半页，翻指定的行数，参见scroll.txt帮助手册页。

[在文件中移动]

vim提供了一些命令，可以方便的在文件中移动。

命令"gg"移动到文件的第一行，而命令"G"则移动到文件的最后一行。

命令"G"前可以加上数字，在这里，数字的含义并不是倍数，而是你打算跳转的行号。例如，你想跳转到文件的第1234行，只需输入"1234G"。

你还可以按百分比来跳转，例如，你想跳到文件的正中间，输入"50%"；如果想跳到75%处，输入"75%"。注意，你必须先输入一个数字，然后输入"%"。如果直接输入"%"，那含义就完全不同了。":help N%"阅读更多细节。

在文件中移动，你可能会迷失自己的位置，这时使用"CTRL-G"命令，查看一下自己位置。这个命令会显示出光标的位置及其它信息。为了避免迷失，你可以打开行号显示；使用":set number"命令后，会在每一行前显示出行号，可以更方便的定位的跳转（:help 'number'）。

[移动到指定字符]

上面的命令都是行间移动(除h, l外)，也就是从当前行移动到另外一行。如果我们想在当前行内快速移动，可以使用f, t, F, T命令。

"f"命令移动到光标右边的指定字符上，例如，"fx"，会把移动到光标右边的第一个'x'字符上。"F"命令则反方向查找，也就是移动到光标左边的指定字符上。

"t"命令和"f"命令的区别在于，它移动到光标右边的指定字符之前。例如，"tx"会移动到光标右边第一个'x'字符的前面。"T"命令是"t"命令的反向版本，它移动到光标右边的指定字符之后。

这四个命令只在当前行中移动光标，光标不会跨越回车换行符。

可以在命令前面使用数字，表示倍数。例如，"3fx"表示移动到光标右边的第3个'x'字符上。

";"命令重复前一次输入的f, t, F, T命令，而","命令会反方向重复前一次输入的f, t, F, T命令。这两个命令前也可以使用数字来表示倍数。

[行首/行尾]

在vim中，移动到行首的命令非常简单，就是"0"，这个是数字0，而不是大写字母O。移动到行尾的命令是"$"。

另外还有一个命令"^"，用它可以移动到行首的第一个非空白字符。

在正则表达式中我们会看到，"^"字符代表行首，而"$"字符代表行尾。可见，vi/vim的按键的安排，的确是别具匠心的。

[按单词移动]

我们知道，英文文档的主体是单词，通常用空白字符(包括空格、制表符和回车换行符)来分隔单词。vim中提供了许多命令来按单词移动。

要根据单词来移动，首先要把文本分隔为一个个独立的单词。vim在对单词进行分隔时，会把'iskeyword'选项中的字符做为单词的组成字符。也就是说，一个单词(word)由'iskeyword'选项中定义的字符构成，它前面、后面的字符不在'iskeyword'选项定义的字符中。例如，如果我们把'iskeyword'选项设置为"a-z,A-Z,48-57,_"，那么"FooBar_123"被做为一个单词，而"FooBar-123"被做为三个单词："FooBar"， "-"和"123"。"a-z,A-Z,48-57,_"中的48-57表示ASCII码表中的数字0-9。

vim中，移动光标到下一个单词的词首，使用命令"w"，移动光标到上一个单词的词首，使用命令"b"；移动光标到下一个单词的结尾，用命令"e"，移动光标到上一个单词的结尾，使用命令"ge"。

上面这些命令都使用'iskeyword'选项中的字符来确定单词的分界，还有几个命令，只把空白字符当做"单词"的分界。当然，这里说的"单词"已经不是传统意义上的单词了，而是由非空白字符

构成一串字串。命令"W"移动光标到下个字串的开始，命令"B"移动到上个字串的开始；命令"E"移动到下个字串的结尾，命令"gE"移动到上个字串的结尾。和上面的命令比较一下，发现什么规律没有？

[H/M/L]

注意：这几个命令是大写的。

使用H/M/L这三个键，可以让光标跳到当前窗口的顶部、中间、和底部，停留在第一个非空字符上。H命令和L命令前也可以加一个数字，但数字的含义不再是倍数，而是指距窗口顶部、底部的行数。例如，"3H"表示光标移动到距窗口顶部第3行的位置；"5L"表示光标移动到距窗口底部5行的位置。

[相对于光标滚屏]

在阅读代码时，有时我们需要根据光标所在的位置滚屏，把光标所在行移动窗口的顶端、中间或底部，这时就可以用到"zt"、"zz"和"zb"。这种滚屏方式相对于翻页来讲，它的好处在于，你能够始终以当前光标位置做为参照，不会出现翻几次页后，发现自己迷失了方向。 ^_^

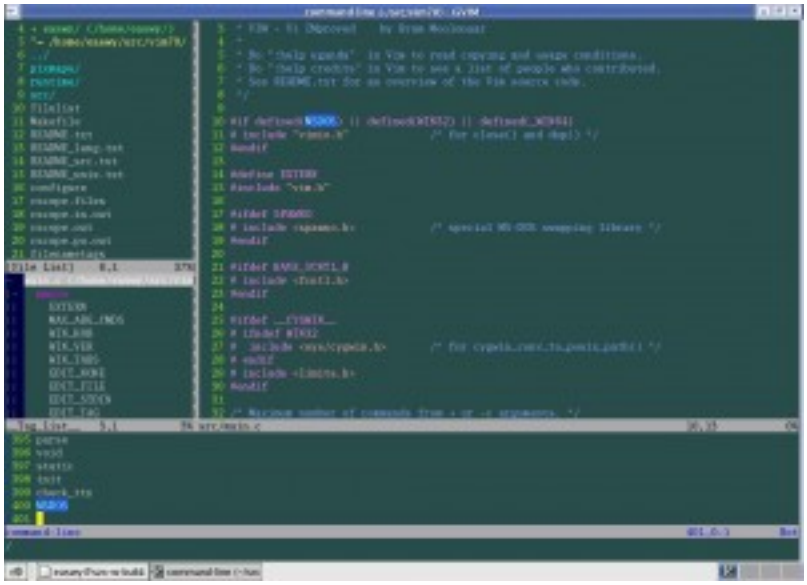
[查找]

查找，也可以做为一种快速移动的方式。

在vim中查找非常容易，直接在Normal模式下输入"/"，然后输入你想查询的字符串，回车，就跳转到第一个匹配的地方了。"/"是向下查找，而"?"进行反方向查找。命令"n"重复上一次的查找命令，而命令"N"也重复上一次的查找命令，只不过它按相反方向查找。

vim保存了查找的历史记录，你可以在输入"/"或"?"后，用上、下光标键(或CTRL-P/CTRL-N)翻看历史记录，然后再次执行这个查找。

另外你还可以使用"q/"和"q?"命令，在vim窗口最下面打开一个新的窗口，这个窗口会列出你的查找历史记录，你可以使用任何vim编辑命令对此窗口的内容进行编辑，然后再按回车，就会对光标所在的行的内容进行查找。



点击查看大图

在上图中，我使用"q/"命令打开了command-line窗口，这个窗口列出了我之前所查找的字符串。我现在想查找包含"check_swap"，于是先跳到第399行，把"check_tty"改为"check_swap"，然后

按回车。此时vim就去查找包含"check_swap"位置了。这个例子比较简单，你可能觉得command-line窗口没什么必要，但如果你要查找的内容是一个很长的正则表达式，你就会发现它非常有用了。

vim中有许多与查找相关的选项设置，其中最常用的是'incsearch', 'hlsearch', 'ignorecase'。

- 'incsearch'表示在你输入查找内容的同时，vim就开始对你输入的内容进行匹配，并显示匹配的位置。打开这个选项，你可以即时看到查找的结果。
- 'hlsearch'选项表示对匹配的所有项目进行高亮显示。
- 'ignorecase'选项表示在查找时忽略大小写。

通常我会打开'incsearch'和'hlsearch'选项，关闭'ignorecase'选项。

下一篇文章介绍了在vim中移动的另外一些方法，这些移动命令的需要的技巧更高一些。

[参考文档]

- vim手册
- vim中文手册

<< 返回vim使用进阶: 目录

第 17 章 指随意动，移动如飞（二）

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

```
:help usr_03.txt
:help motion.txt
:help usr_29.txt
:help scroll.txt
:help folding
```

上一篇文章中我们介绍了一些常用的移动命令，本篇将继续介绍更多的命令，使你在文档中自由穿梭。

[利用跳转表]

在vim中，很多命令可以引起跳转，vim会记住把跳转前光标的位置记录到跳转表中，并提供了一些命令来根据跳转表进行跳转。要知道哪些命令引起跳转，参见":help jump-motions"。

使用命令"''"（两个单引号）和"``"(两个反引号，在键盘上和"~"共用一个键)可以返回到最后跳转的位置。例如，当前光标位于文件中第1234行，然后我使用"4321G"命令跳转到第4321行；这时如果我按"''"或"``"，就会跳回到1234行。

因为这两个命令也属于跳转命令，所以第4321行也被记入跳转表，如果你再次使用这两个命令，就会发现自己又跳回第4321行了。

这两个命令有一点不同，"``"在跳转时会精确到列，而"''"不会回到跳转时光标所在的那一列，而是把光标放在第一个非空白字符上。

如果想回到更老的跳转位置，使用命令"CTRL-O"；与它相对应的，是"CTRL-I"，它跳转到更新的跳转位置(:help CTRL-O和:help CTRL-I)。这两个命令前面可以加数字来表示倍数。

使用命令":jumps"可以查看跳转表(:help :jumps)。

[使用标记]

标记(mark)是vim提供的精确定位技术，其功能相当于GPS技术，只要你知道标记的名字，就可以使用命令直接跳转到该标记所在的位置。

vim中的标记都有一个名字，这个名字用单一的字符表示。大写和小写字母(A-Za-z)都可以做为标记的名字，这些标志的位置可以由用户来设置；而数字标记0-9，以及一些标点符号标记，用户不能进行设置，由vim来自动设置。

我们主要讲述字母标记的使用，对于数字标记和标点符号标记，请自行参阅帮助手册(:help mark-motions)。

小写字母标记局限于缓冲区，也就是说，每个缓冲区都可以定义自己的小写字母标记，各缓冲区间的小写字母标记彼此不干扰。如果我在文件A中设置一个标记t，然后在文件B中也可以设置一个标记t。那么在文件A中，可以用"'t"命令跳到文件A的标记t位置 ；在文件B中，可以用"'t"命令跳到文件B的标记t位置。如果文件在缓冲区列表中被删除，小写字母标记就丢失了。

大写字母标记是全局的，它在文件间都有效。如果在文件A中定义一个标记T，那么当使用命令"'T"时，就会跳到文件A的标记T位置，不管你当前处于哪个文件中。

设定一个标记很简单，使用命令"m{a-zA-Z}"就可以了。例如，命令"mt"在把当前光标位置设定为标记t；命令"mT"把当前光标位置设定为标记T。(:help m)

要跳转到指定的标记，使用命令"'{a-zA-Z}"或"`{a-zA-Z}"。例如，命令"'t"会跳转到标记t；命令"'T"会跳转到标记T。(:help ')

单引号和反引号的区别和上面所讲的一样，"`"在跳转时会精确到列，而"'"不会回到跳转时光标所在的那一列，而是把光标放在第一个非空白字符上。

标记也可以被删除，使用命令":delmarks"可以删除指定标记。命令":marks"列出所有的标记。

关于标记，有两个非常有用的插件，一个是ShowMarks，另外一个叫marks browser。

ShowMarks是我最常用的插件之一，它使用vim提供的sign功能以及高亮功能显示出标记的位置。这样，你在设定了一个标记后，它就会在你的vim窗口中显示出标记的名字，并高亮这一行。

在你的$HOME/.vim目录把它解压，然后进行简单设置。 在我的vimrc中，对ShowMarks进行了如下配置：

```
"""""""""""""""""""""""""""""""
" showmarks setting
"""""""""""""""""""""""""""""""
" Enable ShowMarks
let showmarks_enable = 1
" Show which marks
let showmarks_include = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
" Ignore help, quickfix, non-modifiable buffers
let showmarks_ignore_type = "hqm"
" Hilight lower & upper marks
let showmarks_hlline_lower = 1
let showmarks_hlline_upper = 1
```

首先，使能showmarks插件，然后定义showmarks只显示全部的大写标记和小写，并高亮这两种标记；对文件类型为help、quickfix和不可修改的缓冲区，则不显示标记的位置。

你可以定义自己的颜色来高亮标记所在的行，下面是我的定义，我把它放在我自己的colorscheme文件中：

```
" For showmarks plugin
hi ShowMarksHLl ctermbg=Yellow   ctermfg=Black  guibg=#FFDB72    guifg=Black
hi ShowMarksHLu ctermbg=Magenta  ctermfg=Black  guibg=#FFB3FF    guifg=Black
```

ShowMarks插件中已经定义了一些快捷键：

```
<Leader>mt   - 打开/关闭ShowMarks插件
<Leader>mo   - 强制打开ShowMarks插件
<Leader>mh   - 清除当前行的标记
<Leader>ma   - 清除当前缓冲区中所有的标记
<Leader>mm   - 在当前行打一个标记，使用下一个可用的标记名
```

我最常使用的是"<Leader>mm"和"<Leader>mh"，用起来非常方便。在我的vimrc中，把Leader定义为",", 所以每次都使用",mm"和",mh"来设置和删除mark。

在vim 7.0中，如果大写的标记被定义了，那么函数line()无论在哪个缓冲区里都会返回该标记的行号，导致showmarks在每个缓冲区里都会把这个大写标记显示出来。因此我为这个插件打了个补丁来修正此问题。

vim 7.0中也可以真正的删除一个mark标记，所以也改了showmarks插件的删除标记功能。原来的功能在删除标记时，并未真正删除它，只是把这个标记指向缓冲区的第一行；现在则是真正删除此标记。

如果想使用我为showmarks打的补丁，请点击这里下载showmarks补丁。

用法：

1. 保存该patch到某一目录，例如：/tmp/showmarks.vim.patch

2. cd到你的.vim目录：cd ~/.vim

3. 运行命令：cat /tmp/showmarks.vim.patch | patch -p0

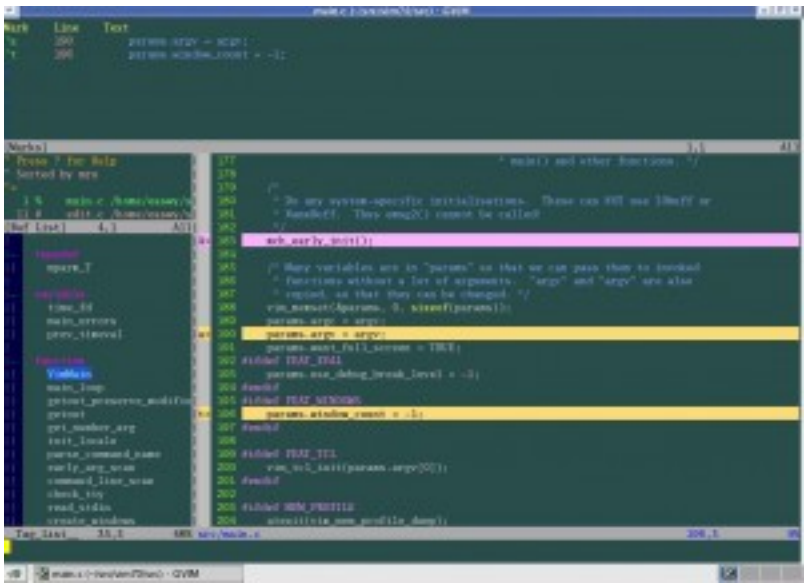
Marks Browser插件可以显示出当前缓冲区中定义的小写标记的位置，在你无法对应上标记的名字和其位置时，非常有用。

下载后把它放到你的$HOME/.vim/plugin目录即可，我为其定义了一个快捷键：

```
"""""""""""""""""""""""""""""""
" markbrowser setting
"""""""""""""""""""""""""""""""
nmap <silent> <leader>mk :MarksBrowser<cr>
```

这样，直接使用",mk"就可以打开Mark Browser窗口了。

下图显示这两个插件工作时的效果。我在文件中定义了三个标记，一个大写标记A，两个小写标记a和t。最上面的窗口是Mark Browser窗口，主编辑窗口中的高亮行及sign标记是ShowMarks插件放置的。



点击查看大图

[折行]

在文件比较大时，在文件中移动也许会比较费力。这个时候，你可以根据自己的需要把暂时不会访问的文本折叠起来，既减少了对空间的占用，移动速度也会快很多。

vim提供了多种方法来进行折叠，既可以手动折叠，也可以根据缩进、语法，或使用表达式来进行折叠。

程序文件一般都具有良好的结构，所以根据语法进行折叠是一个不错的选择。

要启用折叠，首先要使能'foldenable'选项，这个选项是局部于窗口的选项，因此可以为每个窗口定义不同的折叠。

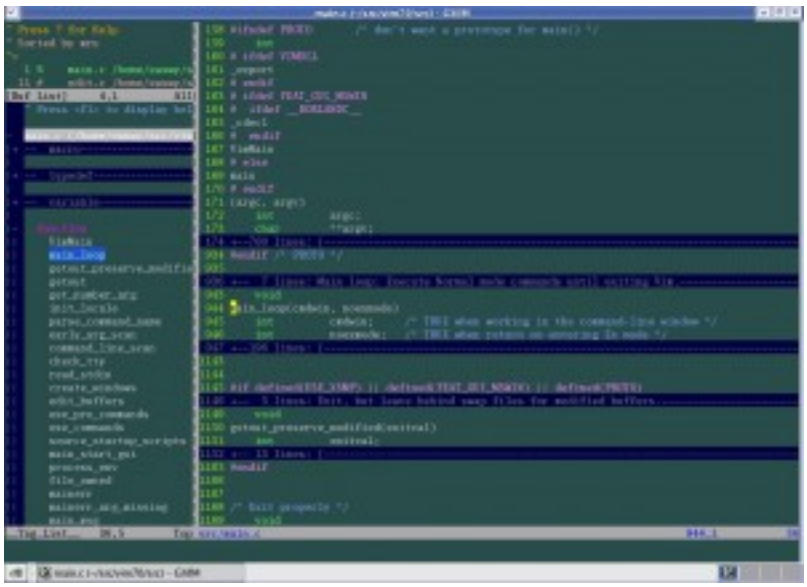
接下来，设置'foldmethod'选项，对于程序，我们可以选择根据语法高亮进行折叠。需注意的，要根据语法高亮进行折叠，必须打开文件类型检测和语法高亮功能，请参见我前面的文章。

下面是我的vimrc中的设置，它使用了自动命令，如果发现文件类型为c或cpp，就启用折叠功能，并按语法进行折叠：

```
autocmd FileType c,cpp  setl fdm=syntax | setl fen
```

注意，vim的很多命令、选项名都有简写形式，在帮助手册中可以看到简写形式，也可以按简写形式来help，例如，要查看'foldmethod'选项的帮助，可以只输入":help 'fdm'"。

折叠后的效果见下图：



点击查看大图

图中以黑色背景显示的行就是被折叠起来的行，vim会显示这个fold中被折叠了多少行，以及起始行的内容。留意一下左下方的"__Tag_List__"窗口，在这个窗口中也存在着折叠，我把macro, typedef, variable几项折叠起来了，而把function的折叠打开。从该窗口最左边的折叠栏(:help fold-foldcolumn)也可以看出不同：被折叠的文本前显示了"+"，打开的折叠前显示的是"|"。

折叠的背景色及显示文字等都可以修改，参阅帮助手册(:help folding)。

下面的命令用来打开和关闭折叠：

zo - 打开光标下的折叠
z0 - 循环打开光标下的折叠，也就是说，如果存在多级折叠，每一级都会被打开
zc - 关闭光标下的折叠
zC - 循环关闭光标下的折叠

更多的命令，请参阅手册(:help folding)。

vim提供了一些命令在折叠间快速移动：

[z - 到当前打开折叠的开始
]z - 到当前打开折叠的结束
zj - 向下移动到下一个折叠的开始处
zk - 向上移动到上一个折叠的结束处

我通常不喜欢把文本折叠起来，因为我更喜欢一目了然的看到全部文本。你可以根据自己的喜好来决定是否启用折叠。

多说一点，手动创建的折叠是可以保存在session文件中的，这样下次进入vim时可以载入之前创建的折叠，参见:help 'sessionoptions'。

[在程序中移动]

vim的作者是一个程序员，这就不难理解为什么vim提供了众多在程序中移动的命令。这里面既包括我们前面的文章中介绍过的利用tag文件及cscope在标签间跳转，也包括众多在函数、注释、预处理指令、程序段，及其它程序元素中移动的命令。

本文不再详细介绍这些命令，作为程序员，一定要熟读usr_29.txt！这些命令，可以帮助你在程序中得心应手的移动。

在这里介绍两个插件，增强了在程序中移动的功能，一个是a.vim，另外一个是matchit。

a.vim的功能非常简单，它帮助你在源文件和头文件间进行切换，这个简单的功能，却非常实用，至少它为我节省了很多时间。

下载a.vim后，把它放到你的.vim/plugin目录就可以了。

假设你正在浏览C语言的源文件，这时想修改它对应的头文件，只需要输入":A"命令，就切换到头文件了（需要源文件和头文件在同一目录中）。a.vim插件还定义了其它一些命令和快捷键，参见它的帮助手册。

在vim中，"%"命令跳转到与当前项目相匹配的项目。例如，当光标位置在"{"时，按下%，光标就跳转到对应的"}"(:help %)。

但vim提供的%命令，只能在括号，或者C注释的开始和结束(/* */)，或者C编译预处理指令间进行跳转。对于其它程序结构，例如HTML，%命令不能从<html>标记，跳转到对应的</html>标记。

Matchit插件则扩展了%命令的功能，使%命令可以对其它程序语言的开始和结束标记间进行跳转。

下载后，把这个插件放到你的.vim/plugin目录，你就可以用%在各种开始/结束标记间跳转了，目前，它可以支持Ada, ASP with VBS, Csh, DTD, Essbase, Fortran, HTML, JSP (same as HTML), LaTeX, Lua, Pascal, SGML, Shell, Tcsh, Vim, XML等语言。

[插入模式下的移动]

上面介绍的移动命令，都是在normal模式下使用的，如果想在insert模式下移动，阅读:help ins-special-special。

你真的需要在插入模式下移动吗？我几乎不会！通常我会先按ESC返回Normal模式，然后再移动，当你习惯了以后，你会发现效率会更高。

[小结]

你会发现，本文的内容，和usr_03.txt帮助文档很相似。是的，只要你学会了usr_03.txt中列出的命令，你就掌握了最常用最实用的vim移动命令(:help usr_03.txt)。

如果你想了解更多的移动命令，请通篇阅读motion.txt，记住你最有可能用到的那些键。当你的手指能够不假思索的使用这些命令后，你在vim中就能做到指随意动、移动如飞了。

[参考文档]

- vim手册
- vim中文手册

<< 返回vim使用进阶: 目录

第 18 章 在vim中使用gdb调试

<< 返回vim使用进阶: 目录

本节所用命令的帮助入口：

```
:help vimgdb
```

在UNIX系统最初设计时，有一个非常重要的思想：每个程序只实现单一的功能，通过管道等方式把多个程序连接起来，使之协同工作，以完成更强大的功能。程序只实现单一功能，一方面降低了程序的复杂性，另一方面，也让它专注于这一功能，把这个功能做到最好。就好像搭积木一样，每个积木只提供简单的功能，但不同的积木垒在一起，就能搭出大厦、汽车等等复杂的东西。

从UNIX系统(及其变种，包括Linux)的命令行就可以看出这一点，每个命令只专注于单一的功能，但通过管道、脚本等把这些命令揉合起来，就能完成复杂的任务。

vi/vim的设计也遵从这一思想，它只提供了文本编辑功能(与Emacs的大而全刚好相反)，而且正如大家所看到的，它在这一领域做的是如此的出色。

也正因为如此，vim自身并不提供集成开发环境所需的全部功能(它也不准备这样做，vim只想成为一个通用的文本编辑器)。它把诸如编译、调试这样功能，交给更专业的工具去实现，而vim只提供与这些工具的接口。

我们在前面已经介绍过vim与编译器的接口(即quickfix)，vim也提供了与调试器的接口，这一接口就是netbeans。除此之外，还可以给vim打一个补丁，以使其支持gdb调试器。

由于netbeans接口只能在gvim中使用，而使用vimgdb补丁，无论在终端的vim，还是gvim，都可以调试。所以我更喜欢打补丁的方式，我首先介绍这种方法。

打补丁的方式，需要重新编译vim，刚好借这个机会，介绍一下vim的编译方法。我只介绍Linux上编译方法，如果你想在windows上编译vim，可以参考这篇文档：Vim: Compiling HowTo: For Windows。

[下载vim源代码]

首先我们需要下载vim的源码。到vim主页下载当前最新的vim 7.1的源代码，假设我们把代码放到~/install/目录，文件名为vim-7.1.tar.bz2。

[下载vimgdb补丁]

接下来，我们需要下载vimgdb补丁，下载页面在：

http://sourceforge.net/project/showfiles.php?group_id=111038&package_id=120238

在这里，选择vim 7.1的补丁，把它保存到~/install/vimgdb71-1.12.tar.gz。

[打补丁]

运行下面的命令，解压源码文件，并打上补丁：

```
cd ~/install/
tar xjf vim-7.1.tar.bz2
tar xzf vimgdb71-1.12.tar.gz
patch -d vim71 --backup -p0 < vimgdb/vim71.diff
```

[定制vim的功能]

缺省的vim配置已经适合大多数人，但有些时候你可能需要一些额外的功能，这时就需要自己定制一下vim。定制vim很简单，进入~/install/vim71/src文件，编辑Makefile文件。这是一个注释很好的文档，根据注释来选择：

- 如果你不想编译gvim，可以打开--disable-gui选项；
- 如果你想把perl, python, tcl, ruby等接口编译进来的话，打开相应的选项，例如，我打开了--enable-tclinterp选项；
- 如果你想在vim中使用cscope的话，打开--enable-cscope选项；
- 我们刚才打的vimgdb补丁，自动在Makefile中加入了--enable-gdb选项；
- 如果你希望在vim使用中文，使能--enable-multibyte和--enable-xim选项；
- 可以通过--with-features=XXX选项来选择所编译的vim特性集，缺省是--with-features=normal；
- 如果你没有root权限，可以把vim装在自己的home目录，这时需要打开prefix = $(HOME)选项；

编辑好此文件后，就可以编辑安装vim了。如果你需要更细致的定制vim，可以修改config.h文件，打开/关闭你想要的特性。

[编译安装]

编译和安装vim非常简单，使用下面两个命令：

```
make
make install
```

你不需要手动运行./configure命令，make命令会自动调用configure命令。

上面的命令执行完后，vim就安装成功了。

我在编译时打开了"prefix = $(HOME)"选项，因此我的vim被安装在~/bin目录。这时需要修改一下PATH变量，以使其找到我编辑好的vim。在~/.bashrc文件中加入下面这两句话：

```
PATH=$HOME/bin:$PATH
export PATH
```

退出再重新登录，现在再敲入vim命令，发现已经运行我们编译的vim了。

[安装vimgdb的runtime文件]

运行下面的命令，解压vimgdb的runtime文件到你的~/.vim/目录：

```
cd ~/install/vimgdb/
tar zxf vimgdb_runtime.tgz -C~/.vim/
```

现在启动vim，在vim中运行下面的命令以生成帮助文件索引：

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

现在，你可以使用":help vimgdb"命令查看vimgdb的帮助了。

至此，我们重新编译了vim，并为之打上了vimgdb补丁。下面我以一个例子来说明如何在vim中完成"编码—编译—调试"一条龙服务。

[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，不过最好使用gdb 6.0以上的版本。

我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。先来看示例代码：

文件~/tmp/sample.c内容如下，这是主程序，调用函数计算某数的阶乘并打印：

```
/* ~/tmp/sample.c */
#include <stdio.h>
extern int factor(int n, int *rt);

int main(int argc, char **argv)
{
    int i;
    int result = 1;

    for (i = 1; i < 6; i++)
    {
        factor(i, &result);
        printf("%d! = %d\n", i, result);
    }

    return 0;
}
```

文件~/tmp/factor/factor.c内容如下，定义了子函数factor()。之所以把它放到子目录factor/，是为了演示vimgdb可以根据调试位置自动打开文件，不管该文件在哪个目录下：

```
/* ~/tmp/factor/factor.c */
int factor(int n, int *r)
{
    if (n <= 1)
        *r =  n;
    else
    {
        factor(n - 1, r);
        *r *= n;
    }

    return 0;
}
```

Makefile文件，用来编译示例代码，最终生成的可执行文件名为sample。

```
# ~/tmp/Makefile
```

```
sample: sample.c factor/factor.c
gcc -g -Wall -o sample sample.c factor/factor.c
```

假设vim的当前工作目录是~/tmp(使用":cd ~/tmp"命令切换到此目录)。我们编辑完上面几个文件后，输入命令":make"，vim就会根据Makefile文件进行编译。如果编译出错，vim会跳到第一个出错的位置，改完后，用":cnext"命令跳到下一个错误，以此类推。这种开发方式被称为quickfix，我们已经在剑不离手 - quickfix一文中讲过，不再赘述。

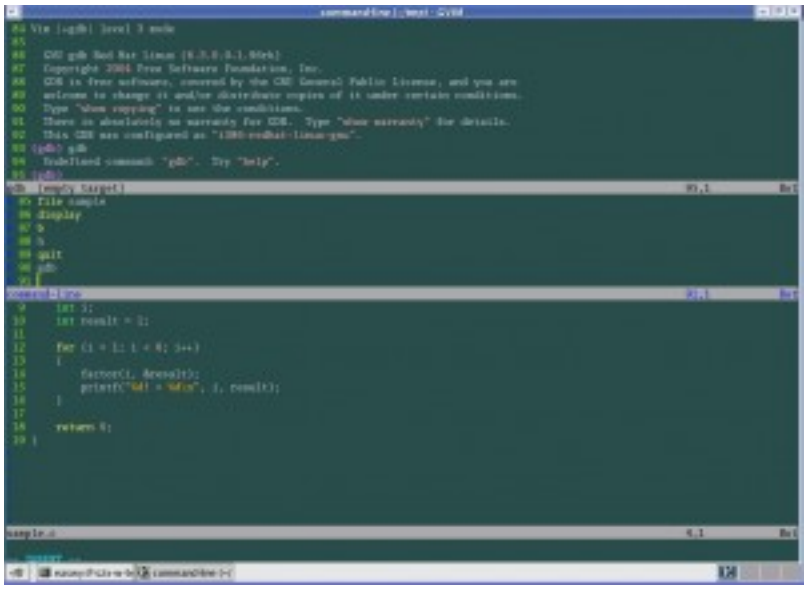
现在，假设已经完成链接，生成了最终的sample文件，我们就可以进行调试了。

vimgdb补丁已经定义了一些键绑定，我们先加载这些绑定：

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

加载后，一些按键就被定义为调试命令(vimgdb定义的键绑定见":help gdb-mappings")。按<F7>可以在按键的缺省定义和调试命令间切换。

好了，我们现在按空格键，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。现在，我们在这个窗口中输入"gdb"，按回车后，command-line窗口自动关闭，而在当前窗口上方又打开一个窗口，这个窗口是gdb输出窗口。现在vim的窗口布局如下(我又按空格打开了command-line窗口)：



点击查看大图

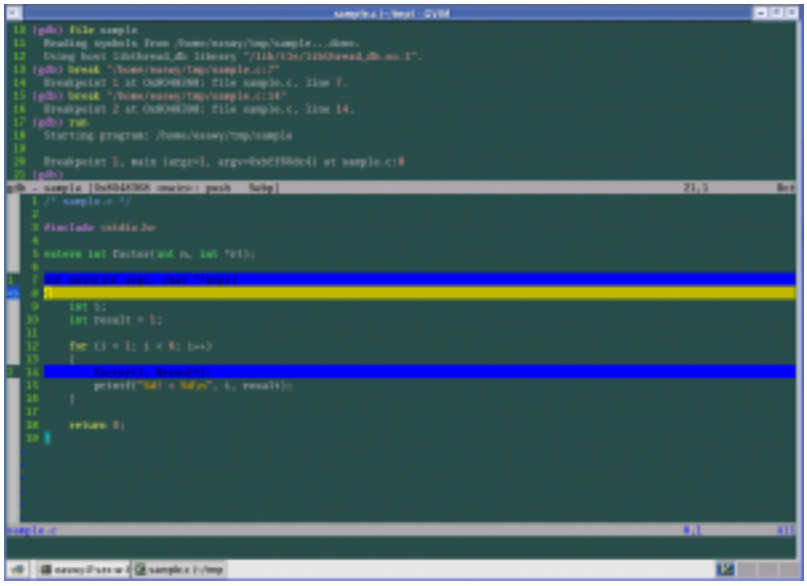
小技巧: command-line窗口是一个特殊的窗口，在这种窗口中，你可以像编辑文本一样编辑命令，完成编辑后，按回车，就会执行此命令。你要重复执行某条命令，可以把光标移到该命令所在的行，然后按回车即可；你也可以对历史命令进行修改后再执行。详见":help cmdline-window"。

接下来，在command-line窗口中输入以下命令：

```
cd ~/tmp
file sample
```

这两条命令切换gdb的当前工作目录，并加载我们编译的sample程序准备调试。

现在使用vim的移动命令，把光标移动到sample.c的第7行和14行，按"CTRL-B"在这两处设置断点，然后按"R"，使gdb运行到我们设置的第一个断点处("CTRL-B"和"R"都是gdb_mappings.vim定义的键绑定，下面介绍的其它调试命令相同)。现在vim看起来是这样：



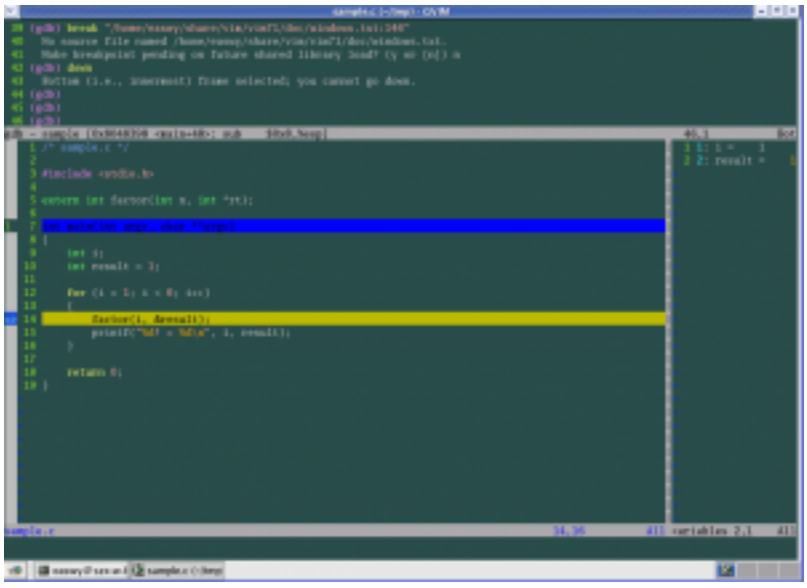
点击查看大图

断点所在的行被置以蓝色，并在行前显示标记1和2表明是第几个断点；程序当前运行到的行被置以黄色，行前以"=>"指示，表明这是程序执行的位置(显示的颜色用户可以调整)。

接下来，我们再按"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的窗口：

```
:bel 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，用同样的方式把变量result也加入到监视窗口，这里可以从监视窗口中看到变量i和result的值。

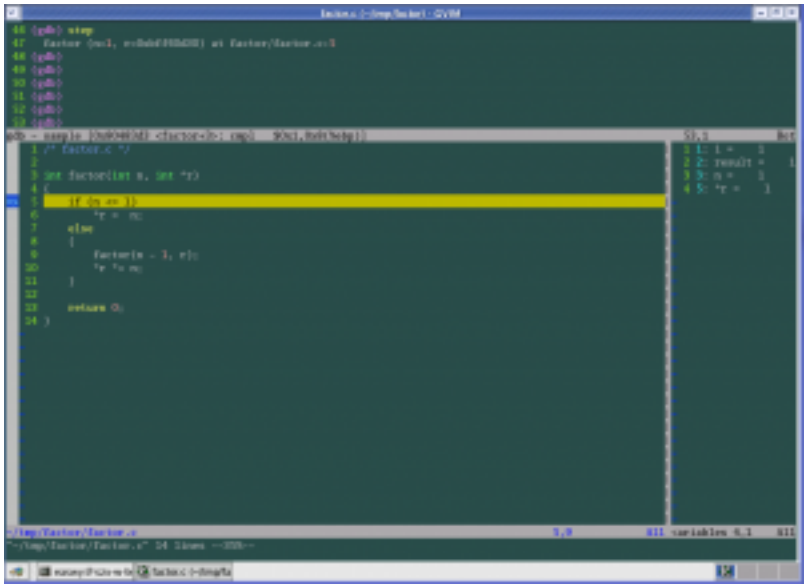


点击查看大图

现在我们按"S"步进到factor函数，vim会自动打开factor/factor.c文件并标明程序执行的位置。我们再把factor()函数中的变量n加入到监视窗口；然后按空格打开command-line窗口，输入下面的命令，把变量*r输入到变量窗口：

```
createvar *r
```

现在，vim看起来是这样的：



点击查看大图

现在，你可以用"S"、"CTRL-N"或"C"来继续执行，直至程序运行结束。

如果你是单步执行到程序结束，那么vim最后可能会打开一个汇编窗口。是的，vimgdb支持汇编级的调试。这里我们不用进行汇编级调试，忽略即可。

如果你发现程序有错误，那么可以按"Q"退出调试(gdb会提示是否退出，回答y即可)，然后修改代码、编译、调试，直到最终完成。在修改代码时，你可能并不喜欢vimgdb的键映射(例如，它把CTRL-B映射为设置断点，而这个键又是常用的翻页功能)，你可以按<F7>取消vimgdb的键映射，或者你直接修改gdb_mappings.vim文件中定义的映射。

看，vim + gdb调试是不是很简单?!

我们可以再定制一下，使调试更加方便。

打开~/.vim/macros/ gdb_mappings.vim文件，在"let s:gdb_k = 0"这一行下面加上这段内容：

```
" easwy add
if ! exists("g:vimgdb_debug_file")
    let g:vimgdb_debug_file = ""
elseif g:vimgdb_debug_file == ""
    call inputsave()
    let g:vimgdb_debug_file = input("File: ", "", "file")
    call inputrestore()
endif
```

```
call gdb("file " . g:vimgdb_debug_file)
" easwy end
```

在"let s:gdb_k = 1"这一行下面加上这段内容：

```
" easwy add
call gdb("quit")
" end easwy
```

注释掉最后一行的"call s:Toggle()"。

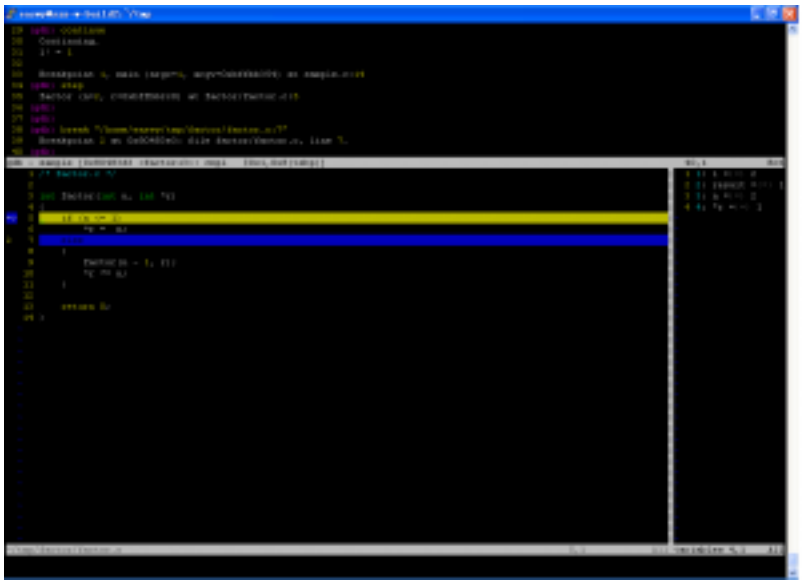
然后在你的vimrc中增加这段内容：

```
""""""""""""""""""""""""""""""""
" vimgdb setting
""""""""""""""""""""""""""""""""
let g:vimgdb_debug_file = ""
run macros/gdb_mappings.vim
```

现在，在启动vim后，按<F7>，就进入调试模式以及设定调试的键映射。在第一次进入调试模式时，会提示你输入要调试的文件名，以后就不必再输入了。再按一次<F7>，就退出调试模式，取消调试的键映射。

利用vim的键映射(map)机制，你可以把你喜欢的gdb命令映射为vim的按键，方便多了。映射的例子可以参照~/.vim/macros/ gdb_mappings.vim。

再附上一张抓图，这是使用putty远程登录到linux上，在终端vim中进行调试。这也是我为什么喜欢vimgdb的原因，因为它可以在终端vim中调试，而clewn只支持gvim：



点击查看大图

因为我不常使用gdb调试，所以本文仅举了个简单的例子，以抛砖引玉。欢迎大家共享自己的经验和心得。

最后，让我们感谢vimgdb作者xdegaye的辛勤劳动，我们后续文章会介绍clewn，这是vim与gdb结合的另外一种形式，它和vimgdb同属一个项目。

[参考文档]

- vim帮助文件
- vim中文手册
- http://clewn.sourceforge.net/index.html

<< 返回vim使用进阶: 目录

第 19 章 vim编译中遇到的问题及解决方法

<< 返回vim使用进阶: 目录

在文章在vim中使用gdb调试中，我介绍了如何编译vim。

不过有网友在编译vim时遇到问题，问如何解决vim编译中出现的问题，我把解决方法总结在这里。

一个比较常见的问题就是在编译gvim不成功。vim编译完了，却发现图形化的gvim没有被编译出来。

vim在编译时，缺省会尝试编译gvim，但如果需要的图形库或其它库文件没有找到，就会略过gvim的编译。

出现这个问题，首先检查你的图形库是否存在。通常我们所用的图形库都是gtk2，如果你的计算机上安装上gnome，那么肯定已经安装了gtk的图形库。如果确是因gtk库没有安装，可以先安装gtk库，网上关于gtk安装的文章很多，在此不再赘述了。

如果你的计算机已经安装了gtk2，但gvim还是编译失败，就需要查看一下configure的输出，看看为什么不能编译gvim。configure的输出为vim72/src/auto/config.log。

例如，在我的debian计算机上，编译gvim失败，在config.log中，可以看到：

```
1. configure:7601: checking if X11 header files can be found
2. configure:7627: gcc -c -O2 -fno-strength-reduce -Wall conftest.c >&5
3. conftest.c:16:27: error: X11/Intrinsic.h: No such file or directory
4. configure:7634: $? = 1
5. configure: failed program was:
6. | /* confdefs.h. */
7. | #define PACKAGE_NAME “”
8. | #define PACKAGE_TARNAME “”
9. | #define PACKAGE_VERSION “”
10. | #define PACKAGE_STRING “”
11. | #define PACKAGE_BUGREPORT “”
12. | #define UNIX 1
13. | #define STDC_HEADERS 1
14. | #define HAVE_SYS_WAIT_H 1
15. | #define FEAT_NORMAL 1
```

```
16. | #define USE_XSMP_INTERACT 1
17. | #define HAVE_LIBNSL 1
18. | #define FEAT_NETBEANS_INTG 1
19. | /* end confdefs.h. */
20. | #include <X11/Xlib.h>
21. | #include <X11/Intrinsic.h>
22. | int
23. | main ()
24. | {
25. |
26. | ;
27. | return 0;
28. | }
29. configure:7645: result: no
30. configure:7978: checking  - enable-gui argument
31. configure:8038: result: no GUI support
```

我们可以看到在第30行和31行显示没有GUI支持，而原因则在第3行："conftest.c:16:27: error: X11/Intrinsic.h: No such file or directory"，也就是说找不到文件 X11/Intrinsic.h。

在网上搜索后得知，这个库在debian的libdevel/libxt-dev包中，安装此包后，gvim就编译成功了。

另外还有一个朋友问，如何使vim支持+signs功能，signs功能是在big版本中才被包含进来的功能，如果想在normal版本的vim中包含此功能，就需要修改vim72/src/feature.h文件，在此文件中把：

```
# define FEAT_SIGNS
```

打开，然后再编译就可以了。

<< 返回vim使用进阶: 目录