

Vim 8 下 C/C++ 开发环境搭建

 April 22nd, 2018  skywind

挺多人问怎么在 Vim 中搭建 C/C++ 开发环境，我本来想找篇文章发给人家，结果网上看了一圈，要不就是内容太过陈旧，要不就是太过零碎，不成体系。2018 年了，Vim 8 发布已经一年半，各大 Linux 发行版和 Mac OS X 自带的 Vim 都已经跟进到 8 了，不少文章还在介绍一些十年前的老方法。于是有了这篇文章。

那如何高效的再 Vim 8 中开发 C/C++ 项目呢？假设你已经有一定 Vim 使用经验，并且折腾过 Vim 配置，能够相对舒适的在 Vim 中编写其他代码的时候，准备在 Vim 开始 C/C++ 项目开发，或者你已经用 Vim 编写了几年 C/C++ 代码，想要更进一步，让自己的工作更加顺畅的话，本文就是为你准备的：

插件管理

为什么把插件管理放在第一个来讲呢？这是比较基本的一个东西，如今 Vim 下熟练开发的人，基本上手都有 20-50 个插件，遥想十年前，Vim 里常用的插件一只手都数得过来。过去我一直使用老牌的 [Vundle](#) 来管理插件，但是随着插件越来越多，更新越来越频繁，Vundle 这种每次更新就要好几分钟的东西实在是不堪重负了，在我逐步对 Vundle 失去耐心之后，我试用了 [vim-plug](#)，用了两天以后就再也回不去 Vundle 了，它支持全异步的插件安装，安装 50 个插件只需要一分钟不到的时间，这在 Vundle 下面根本不可想像的事情，插件更新也很快，不像原来每次更新都可以去喝杯茶去，最重要的是它支持插件延迟加载：

```
1 | " 定义插件，默认用法，和 Vundle 的语法差不多
2 | Plug 'junegunn/vim-easy-align'
3 | Plug 'skywind3000/quickmenu.vim'
4 |
5 | " 延迟按需加载，使用到命令的时候再加载或者打开对应文件类型才加载
6 | Plug 'scrooloose/nerdtree', { 'on': 'NERDTreeToggle' }
7 | Plug 'tpope/vim-fireplace', { 'for': 'clojure' }
8 |
9 | " 确定插件仓库中的分支或者 tag
10 | Plug 'rdnetto/YCM-Generator', { 'branch': 'stable' }
11 | Plug 'nsf/gocode', { 'tag': 'v.20150303', 'rtp': 'vim' }
```

定义好插件以后一个：`:PlugInstall` 命令就并行安装所有插件了，比 Vundle 快捷不少，关键是 vim-plug 只有单个文件，正好可以放在我 github 上的 vim 配置仓库中，每次需要更新 vim-plug 时只需要 `:PlugUpgrade`，即可自我更新。

抛弃 Vundle 切换到 vim-plug 以后，不仅插件安装和更新快了一个数量级，大量的插件我都配置成了延迟加载，Vim 启动速度比 Vundle 时候提高了不少。使用 Vundle 的时候一旦插件数量超过 30 个，管理是一件很痛苦的事情，而用了 vim-plug 以后，50-60 个插件都轻轻松松。

符号索引

现在有好多 ctags 的代替品，比如 gtags, etags 和 cquery。然而我并不排斥 ctags，因为他支持 50+ 种语言，没有任何一个符号索引工具有它支持的语言多。同时 Vim 和 ctags 集成的相当好，用它依赖最少，大量基础工作可以直接通过 ctags 进行，然而到现在为止，我就没见过几个人把 ctags 用对了的。

就连配置文件他们都没写对，正确的 ctags 配置应该是：

```
1 | set tags=./.tags;,.tags
```

这里解释一下，首先我把 tag 文件的名字从 tags 换成了 .tags，前面多加了一个点，这样即便放到项目中也不容易污染当前项目的文件，删除时也好删除，gitignore 也好写，默认忽略点开头的文件名即可。

前半部分 `./.`tags; 代表在文件的所在目录下（不是 `:pwd` 返回的 Vim 当前目录）查找名字为 .tags 的符号文件，后面一个分号代表查找不到的话向上递归到父目录，直到找到 .tags 文件或者递归到了根目录还没找到，这样对于复杂工程很友好，源代码都是分布在不同子目录中，而只需要在项目顶层目录放一个 .tags 文件即可；逗号分隔的后半部分 .tags 是指同时在 Vim 的当前目录（`:pwd` 命令返回的目录，可以用 `:cd ..` 命令改变）下面查找 .tags 文件。

最后请更新你的 ctags，不要再使用老旧的 Exuberant Ctags，这货停止更新快十年了，请使用最新的 [Universal CTags](#) 代替之，它在 Exuberant Ctags 的基础上继续更新迭代了近十年，如今任然活跃的维护着，功能更强大，语言支持更多。

（注意最新版 universal ctags 调用时需要加一个 `-output-format=e-ctags`，输出格式才和老的 exuberant ctags 兼容否则会有 windows 下路径名等小问题）。

自动索引

过去写几行代码又需要运行一下 ctags 来生成索引，每次生成耗费不少时间。如今 Vim 8 下面自动异步生成 tags 的工具有很多，这里推荐最好的一个：vim-gutentags，这个插件主要做两件事情：

- 确定文件所属的工程目录，即文件当前路径向上递归查找是否有 .git, .svn, .project 等标志性文件（可以自定义）来确定当前文档所属的工程目录。
- 检测同一个工程下面的文件改动，能会自动增量更新对应工程的 .tags 文件。每次改了几行不用全部重新生成，并且这个增量更新能够保证 .tags 文件的符号排序，方便 Vim 中用二分查找快速搜索符号。

vim-gutentags 需要简单配置一下：

```
1 | " gutentags 搜索工程目录的标志，碰到这些文件/目录名就停止向上一级目录递归
2 | let g:gutentags_project_root = ['.root', '.svn', '.git', '.hg', '.project']
3 |
4 | " 所生成的数据文件的名称
5 | let g:gutentags_ctags_tagfile = '.tags'
6 |
7 | " 将自动生成的 tags 文件全部放入 ~/.cache/tags 目录中，避免污染工程目录
8 | let s:vim_tags = expand('~/.cache/tags')
9 | let g:gutentags_cache_dir = s:vim_tags
10 |
11 | " 配置 ctags 的参数
12 | let g:gutentags_ctags_extra_args = ['--fields=+niasS', '--extra=+q']
13 | let g:gutentags_ctags_extra_args += ['--c++-kinds=+px']
14 | let g:gutentags_ctags_extra_args += ['--c-kinds=+px']
```

有了上面的设置，你平时基本感觉不到 tags 文件的生成过程了，只要文件修改过，gutentags 都在后台为你默默打点是否需要更新数据文件，你根本不用管，还会帮你：

```
1 | setlocal tags+=...
```

为当前文件添加上对应的 tags 文件的路劲而不影响其他文件。得益于 Vim 8 的异步机制，你可以任意随时使用 ctags 相关功能，并且数据库都是最新的。需要注意的是，gutentags 需要靠上面定义的 project_root 里的标志，判断文件所在的工程，如果一个文件没有托管在 .git/.svn 中，gutentags 找不到工程目录的话，就不会为该野文件生成 tags，这也很合理。想要避免的话，你可以在你的野文件目录中放一个名字为 .root 的空白文件，主动告诉 gutentags 这里就是工程目录。

最后啰嗦两句，少用 CTRL-] 直接在当前窗口里跳转到定义，多使用 CTRL-W] 用新窗口打开并查看光标下符号的定义，或者 CTRL-W } 使用 preview 窗口预览光标下符号的定义。

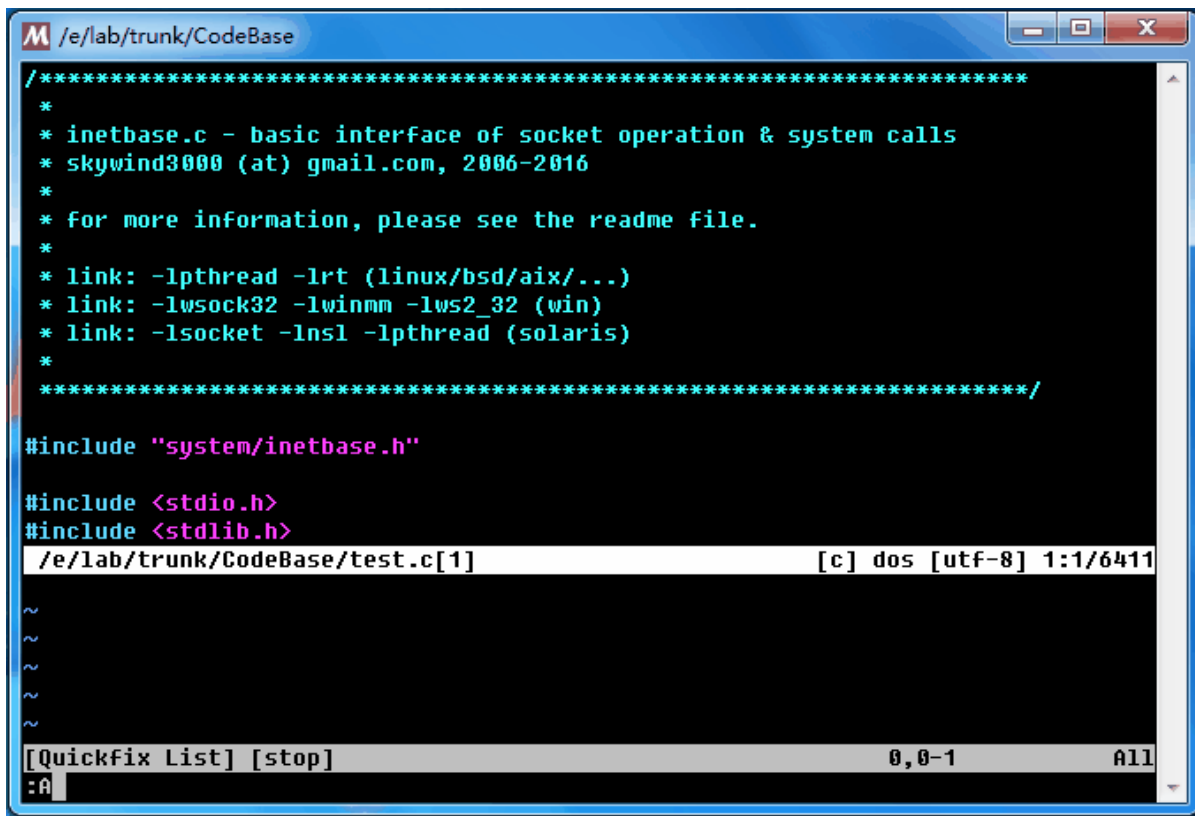
我自己还写过不少关于 ctags 的 vimscript，例如在最下面命令行显示函数的原型而不用急着跳转，或者重复按 ALT+; 在 preview 窗口中轮流查看多个定义，不切走当前窗口，不会出一个很长的列表让你选择，有兴趣可以创我的 vim dotfiles。

编译运行

再 Vim 8 以前，编译和运行程序要么就让 vim 傻等着结束，不能做其他事情，要么切到一个新的终端下面去单独运行编译命令和执行命令，要么开个 tmux 左右切换。如今新版本的异步模式可以让这个流程更加简化，这里我们使用 AsyncRun 插件，简单设置下：

```
1 | Plug 'skywind3000/asyncrun.vim'
2 |
3 | " 自动打开 quickfix window，高度为 6
4 | let g:asyncrun_open = 6
5 |
6 | " 任务结束时候响铃提醒
7 | let g:asyncrun_bell = 1
8 |
9 | " 设置 F10 打开/关闭 Quickfix 窗口
10 | nnoremap <F10> :call asyncrun#quickfix_toggle(6)<cr>
```

该插件可以在后台运行 shell 命令，并且把结果输出到 quickfix 窗口：



```

M /e/lab/trunk/CodeBase
/*****
 *
 * inetbase.c - basic interface of socket operation & system calls
 * skywind3000 (at) gmail.com, 2006-2016
 *
 * for more information, please see the readme file.
 *
 * link: -lpthread -lrt (linux/bsd/aix/...)
 * link: -lwsock32 -lwinmm -ws2_32 (win)
 * link: -lsocket -lssl -lpthread (solaris)
 *
 *****/

#include "system/inetbase.h"

#include <stdio.h>
#include <stdlib.h>

/e/lab/trunk/CodeBase/test.c[1] [c] dos [utf-8] 1:1/6411

~
~
~
~
~
[Quickfix List] [stop] 0,0-1 A11
:~

```

最简单的编译单个文件，和 sublime 的默认 build system 差不多，我们定义 F9 为编译单文件：

```
1 | noremap <silent> <F9> :AsyncRun gcc -Wall -O2 "${VIM_FILEPATH}" -o "${VIM_FILEDIR}/${VIM_FILE" <
```

其中 \$(...) 形式的宏在执行时会被替换成实际的文件名或者文件目录

这样按 F9 就可以编译当前文件，同时按 F5 运行：

```
1 | noremap <silent> <F5> :AsyncRun -raw -cwd=${VIM_FILEDIR} "${VIM_FILEDIR}/${VIM_FILENOEXT}" <
```

用双引号引起来避免文件名包含空格，-cwd=\${VIM_FILEDIR} 的意思时在文件文件的所在目录运行可执行，后面可执行使用了全路径，避免 linux 下面当前路径加 ./ 而 windows 不需要的跨平台问题。

参数 -raw 表示输出不用匹配错误检测模板 (errorformat)，直接原始内容输出到 quickfix 窗口。这样你可以一边编辑一边 F9 编译，出错了可以在 quickfix 窗口中按回车直接跳转到错误的位置，编译正确就接着执行。

接下来是项目的编译，不管你直接使用 make 还是 cmake，都是对一群文件做点什么，都需要定位到文件所属项目的目录，AsyncRun 识别当前文件的项目目录方式和 gutentags 相同，从文件所在目录向上递归，直到找到名为 .git, .svn, .hg 或者 .root 文件或者目录，如果递归到根目录还没找到，那么文件所在目录就被当作项目目录，你重新定义项目标志：

```
1 | let g:asyncrun_rootmarks = ['.svn', '.git', '.root', '_darcs', 'build.xml']
```

然后在 AsyncRun 命令行中，用 <root> 或者 \${VIM_ROOT} 来表示项目所在路径，于是我们可以定义按 F7 编译整个项目：

```
1 | noremap <silent> <F7> :AsyncRun -cwd=<root> make <cr>
```

那么如果你有一个项目不在 svn 也不在 git 中怎么查找 <root> 呢？很简单，放一个空的 .root 文件到你的项目目录下就行了，前面配置过，识别名为 .root 的文件。继续配置用 F8 运行当前项目：

```
1 | noremap <silent> <F8> :AsyncRun -cwd=<root> -raw make run <cr>
```

当然，你的 makefile 中需要定义怎么 run，接着按 F6 执行测试：

```
1 | noremap <silent> <F6> :AsyncRun -cwd=<root> -raw make test <cr>
```

如果你使用了 cmake 的话，还可以照葫芦画瓢，定义 F4 为更新 Makefile 文件，如果不用 cmake 可以忽略：

```
1 | noremap <silent> <F4> :AsyncRun -cwd=<root> cmake . <cr>
```

由于 C/C++ 标准库的实现方式是发现在后台运行时会缓存标准输出直到程序退出，你想实时看到 printf 输出的话需要 fflush(stdout) 一下，或者程序开头关闭缓存：setbuf(stdout, NULL); 即可。

同时，如果你开发 C++ 程序使用 std::cout 的话，后面直接加一个 std::endl 就强制刷新缓存了，不需要弄其他。而如果你在 Windows 下使用 GVim 的话，可以弹出新的 cmd.exe 窗口来运行刚才的程序：

```
1 | noremap <silent> <F5> :AsyncRun -cwd=$(VIM_FILEDIR) -mode=4 "$(VIM_FILEDIR)/$(VIM_FILENOEXT)
2 | noremap <silent> <F8> :AsyncRun -cwd=<root> -mode=4 make run <cr>
```

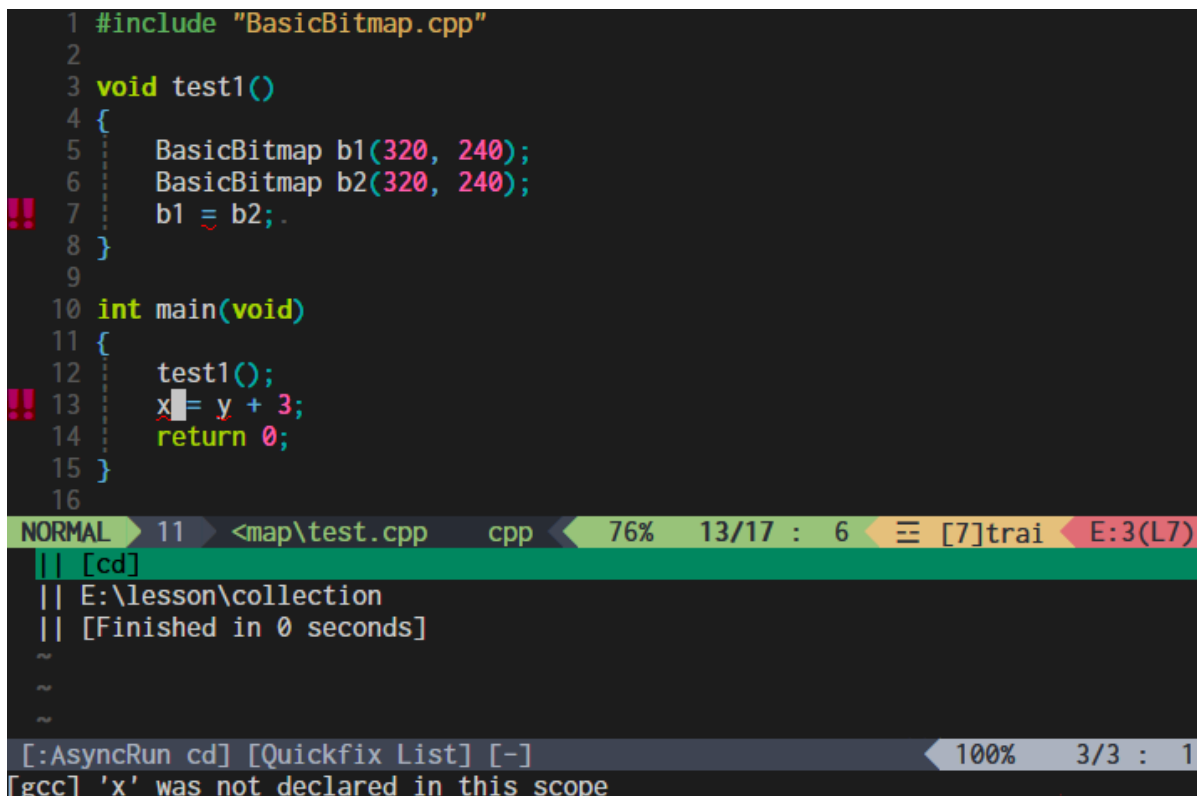
在 Windows 下使用 -mode=4 选项可以跟 Visual Studio 执行命令行工具一样，弹出一个新的 cmd.exe 窗口来运行程序或者项目，于是我们有了下面的快捷键：

- F4: 使用 cmake 生成 Makefile
- F5: 单文件：运行
- F6: 项目：测试
- F7: 项目：编译
- F8: 项目：运行
- F9: 单文件：编译
- F10: 打开/关闭底部的 quickfix 窗口

恩，编译和运行基本和 NotePad++ / GEdit 的体验差不多了。如果你重度使用 cmake 的话，你还可以写点小脚本，将 F4 和 F7 的功能合并，检测 CMakeLists.txt 文件改变的话先执行 cmake 更新一下 Makefile，然后再执行 make，否则直接执行 make，这样更自动化些。

动态检查

代码检查是个好东西，让你在编辑文字的同时就帮你把潜在错误标注出来，不用等到编译或者运行了才发现。我很奇怪 2018 年了，为啥网上还在到处介绍老旧的 syntastic，但凡见到介绍这个插件的文章基本都可以不看了。老的 syntastic 基本没法用，不能实时检查，一保存文件就运行检查器并且等待半天，所以请用实时 linting 工具 ALE：



```
1 #include "BasicBitmap.cpp"
2
3 void test1()
4 {
5     BasicBitmap b1(320, 240);
6     BasicBitmap b2(320, 240);
7     b1 = b2;
8 }
9
10 int main(void)
11 {
12     test1();
13     x = y + 3;
14     return 0;
15 }
16
```

NORMAL 11 <map\test.cpp cpp 76% 13/17 : 6 [7]traï E:3(L7)

|| [cd]
|| E:\lesson\collection
|| [Finished in 0 seconds]
~
~
~

[:AsyncRun cd] [Quickfix List] [-] 100% 3/3 : 1
[gcc] 'x' was not declared in this scope

大概长这个样子，随着你不断的编辑新代码，有语法错误的地方会实时帮你标注出来，侧边会标注本行有错，光标移动过去的时候下面会显示错误原因，而具体错误的符号下面会有红色波浪线提醒。Ale 支持多种语言的各种代码分析器，就 C/C++ 而言，就支持：gcc, clang, cppcheck 以及 clang-format 等，需要另行安装并放入 PATH 下面，ALE 能在你修改了文本后自动调用这些 linter 来分析最新代码，然后将各种 linter 的结果进行汇总并显示再界面上。

同样，我们也需要简单配置一下：

```
1 | let g:ale_linters_explicit = 1
- |
```

```

2 | let g:ale_completion_delay = 500
3 | let g:ale_echo_delay = 20
4 | let g:ale_lint_delay = 500
5 | let g:ale_echo_msg_format = '[%linter%] %code: %s'
6 | let g:ale_lint_on_text_changed = 'normal'
7 | let g:ale_lint_on_insert_leave = 1
8 | let g:airline#extensions#ale#enabled = 1
9
10 | let g:ale_c_gcc_options = '-Wall -O2 -std=c99'
11 | let g:ale_cpp_gcc_options = '-Wall -O2 -std=c++14'
12 | let g:ale_c_cppcheck_options = ''
13 | let g:ale_cpp_cppcheck_options = ''

```

基本上就是定义了一下运行规则，信息显示格式以及几个 linter 的运行参数，其中 6, 7 两行比较重要，它规定了如果 normal 模式下文字改变以及离开 insert 模式的时候运行 linter，这是相对保守的做法，如果没有的话，会导致 YouCompleteMe 的补全对话框频繁刷新。

记得设置一下各个 linter 的参数，忽略一些你觉得没问题的规则，不然没法看。默认错误和警告的风格都太难看了，你需要修改一下，比如我使用 GVim，就重新定义了警告和错误的样式，去除默认难看的红色背景，代码正文使用干净的波浪下划线表示：

```

1 | let g:ale_sign_error = "\ue009\ue009"
2 | hi! clear SpellBad
3 | hi! clear SpellCap
4 | hi! clear SpellRare
5 | hi! SpellBad gui=undercurl guisp=red
6 | hi! SpellCap gui=undercurl guisp=blue
7 | hi! SpellRare gui=undercurl guisp=magenta

```

不同项目之间如果评测标准不一样还可以具体单独制定 linter 的参数，具体见 ALE 帮助文档了。我基本使用两个检查器：gcc 和 cppcheck，都可以在 ALE 中进行详细配置，前者主要检查有无语法错误，后者主要会给出一些编码建议，和对危险写法的警告。

我之前用 syntastic 时就用了两天就彻底删除了，而开始用 ALE 后，一用上就停不下来，头两天我还一度觉得它就是个可有可无的点缀，但是第三天它帮我找出两个潜在的 bug 的时候，我开始觉得没白安装，比如：

```

1 | #include <stdio.h>
2 | #include <errno.h>
3
4 | int main(void)
5 | {
6 |     FILE *fp = fopen("d:/test.txt", "w");
7 |     if (fp == NULL) {
8 |         printf("file is not writable, errno is %d\n", errno);
9 |         return 1;
10 |    }
11 |    else {
12 |        printf("file is writable\n");
13 |    }
14 |    return 0;
15 | }

```

~
~
~
~
~
~
~

NORMAL 23 d:\temp\vim\filewrite.c c 93% 14/15 : 1 E:1(L14)
[cppcheck] Resource leak: fp

即便你使用各类 C/C++ IDE，也只能给实时你标注一些编译错误或者警告，而使用 ALE + cppcheck/gcc，连上面类似的潜在错误都能帮你自动找出来，并且当你光标移动过去时在最下面命令行提示你错误原因。

用上一段时间以后，让你写 C/C++ 有一种安心和舒适的感觉。

修改比较

这是个小功能，在侧边栏显示一个修改状态，对比当前文本和 git/svn 仓库里的版本，在侧边栏显示修改情况，以前 Vim 做不到实时显示修改状态，如今推荐使用 [vim-signify](#) 来实时显示修改状态，它比 gitgutter 强，除了 git 外还支持 svn/mercurial/cvs 等十多

种主流版本管理系统。

没注意到它时，你可能觉得它不存在，当你有时真的看上两眼时，你会发现这个功能很贴心。最新版 signify 还有一个命令:SignifyDiff，可以左右分屏对比提交前后记录，比你命令行 svn/git diff 半天直观多了。并且对我这种同时工作在 subversion 和 git 环境下的情况契合的比较好。

Signify 和前面的 ALE 都会在侧边栏显示一些标记，默认侧边栏会自动隐藏，有内容才会显示，不喜欢侧边栏时有时无的行为可设置强制显示侧边栏：set signcolumn=yes。

文本对象

相信大家用 Vim 进行编辑时都很喜欢文本对象这个概念，diw 删除光标所在单词，ciw 改写单词，vip 选中段落等，ci"/ci(改写引号/括号中的内容。而编写 C/C++ 代码时我推荐大家补充几个十分有用的文本对象，我使用 textobj-user 全家桶：

```
1 Plug 'kana/vim-textobj-user'
2 Plug 'kana/vim-textobj-indent'
3 Plug 'kana/vim-textobj-syntax'
4 Plug 'kana/vim-textobj-function', { 'for':['c', 'cpp', 'vim', 'java'] }
5 Plug 'sgur/vim-textobj-parameter'
```

它新定义的文本对象主要有：

- i, 和 a, : 参数对象，写代码一半在修改，现在可以用 di, / ci, 一次性删除/改写当前参数
- ii 和 ai : 缩进对象，同一个缩进层次的代码，可以用 vii 选中，dii / cii 删除或改写
- if 和 af : 函数对象，可以用 vif / dif / cif 来选中/删除/改写函数的内容

最开始我不太想用额外的文本对象，一直在坚持 Vim 固有的几个默认对象，生怕手练习惯了肌肉形成记忆到远端没有环境的 vim 下形成依赖改不过来，后来我慢慢发现挺有用的，比如改写参数，以前是比较麻烦的事情，这下流畅了很多，当我发现自己编码效率得到比较大的提升时，才发现习惯依赖不重要，行云流水才是真重要。以前看到过无数次都选择性忽略的东西，有时候试试可能会有新的发现。

编辑辅助

大家都知道 color 文件定义了众多不同语法元素的色彩，还有一个关键因素就是语法文件本身能否识别并标记得出众多不同的内容来？语法文件对某些东西没标注，你 color 文件确定了颜色也没用。因此 Vim 下面写 C/C++ 代码，语法高亮准确丰富的话能让你编码的心情好很多，这里推荐 vim-cpp-enhanced-highlight 插件，提供比 Vim 自带语法文件更好的 C/C++ 语法标注，支持 cpp11/14/17。

前面编译运行时需要频繁的操作 quickfix 窗口，ale查错时也需要快速再错误间跳转（location list），就连文件比较也会用到快速跳转到上/下一个差异处，unimpaired 插件帮你定义了一系列方括号开头的快捷键，被称为官方 Vim 中丢失的快捷键。

我们好些地方用到了 quickfix / location 窗口，你在 quickfix 中回车选中一条错误的话，默认会把你当前窗口给切走，变成新文件，虽然按 CTRL+O 可以返回，但是如果不太喜欢这样切走当前文件的做法，可以设置 switchbuf，发现文件已在 Vim 中打开就跳过去，没打开过就新建窗口/标签打开，具体见帮助。

Vim最爽的地方是把所有 ALT 键映射全部留给用户了，尽量使用 Vim 的 ALT键映射，可以让冗长的快捷键缩短很多，请参考：《Vim 和终端软件中支持ALT映射》。

代码补全

传统的 Vim 代码补全基本以 omni 系列补全和符号补全为主，omni 补全系统是 Vim 自带的针对不同文件类型编写不同的补全函数的基础语义补全系统，搭配 neocomplete 可以很方便的对所有补全结果（omni补全/符号补全/字典补全）进行一个合成并且自动弹出补全框，虽然赶不上 IDE 的补全，但是已经比大部分编辑器补全好用很多了。然而传统 Vim 补全还是有两个迈不过去的坎：语义补全太弱，其次是补全分析无法再后台运行，对大项目而言，某些复杂符号的补全会拖慢你的打字速度。

新一代的 Vim 补全系统，YouCompleteMe 和 Deoplete，都支持异步补全和基于 clang 的语义补全，前者集成度高，后者扩展方便。对于 C/C++ 的话，我推荐 YCM，因为 deoplete 的 clang 补全插件不够稳定，太吃内存，并且反应比较慢。所以 C/C++ 的补全的话，请直接使用 YCM，没有之一，而使用 YCM的话，需要进行一些简单的调教：

```
1 let g:ycm_add_preview_to_completeopt = 0
2 let g:ycm_show_diagnostics_ui = 0
3 let g:ycm_server_log_level = 'info'
4 let g:ycm_min_num_identifier_candidate_chars = 2
5 let g:ycm_collect_identifiers_from_comments_and_strings = 1
6 let g:ycm_complete_in_strings=1
7 let g:ycm_key_invoke_completion = '<c-z>'
```

```

8 | set completeopt=menu,menuone
9 |
10 | noremap <c-z> <NOP>
11 |
12 | let g:ycm_semantic_triggers = {
13 |     \ 'c,cpp,python,java,go,erlang,perl': ['re!\w{2}'],
14 |     \ 'cs,lua,javascript': ['re!\w{2}'],
15 |     \ }

```

这样可以输入两个字符就自动弹出语义补全，不用等到 . 或者 → 才触发，同时关闭了预览窗口和代码诊断这些 YCM 花边功能，保持清静，对于原型预览和诊断我们后面有更好的解决方法，YCM这两项功能干扰太大。

上面这几行配置具体每行的含义，可以见：《YouCompleteMe 中容易忽略的配置》。另外我在 Windows 下编译了一个版本，你用 Windows 的话无需下载VS编译，点击 [这里](#)。我日常开发使用 YCM 辅助编写 C/C++，Python 和 Go 代码，基本能提供 IDE 级别的补全。

函数列表

不再建议使用 tagbar，它会在你保存文件的时候以同步等待的方式运行 ctags（即便你没有打开 tagbar），导致vim操作变卡，特别是 windows下开了反病毒软件扫描的话，有时候保存文件卡5-6秒。2018年了，我们有更好的选择，比如使用国人开发的 [LeaderF](#) 来显示函数列表：

```

135 |         goto process_precision;
136 |
137 |     case '.':
138 |         if (width < 0)
139 |             width = 0;
140 |         goto reswitch;
141 |
142 |     case '#':
143 |         altflag = 1;
144 |         goto reswitch;
145 |
146 |     process_precision:
147 |         if (width < 0)
148 |             width = precision, precision = -1;
149 |         goto reswitch;
150 |
151 |     // long flag (doubled for long long)

```

printfmt.c c utf-8[dos] 50% 149/295 : 26

```

1 f printnum(void (*putch)(int, void*), void *putdat, [E:\Lesson\kitus\lib\printfmt>
2 f getuint(va_list *ap, int lflag) [E:\Lesson\kitus\lib\printfmt.c:54 18]
3 f getint(va_list *ap, int lflag) [E:\Lesson\kitus\lib\printfmt.c:67 18]
4 p void printfmt(void (*putch)(int, void*), void *putdat, const char *fmt, ...); [>
5 f void vprintfmt(void (*putch)(int, void*), void *putdat, const char *fmt, va_list >
6 f printfmt(void (*putch)(int, void*), void *putdat, const char *fmt, ...) [E:\Lesso>
7 f static void sprintputch(int ch, struct sprintbuf *b) [E:\Lesson\kitus\lib\prin>
8 f int vsnprintf(char *buf, int n, const char *fmt, va_list ap) [E:\Lesson\kitus\>

```

LeaderF Function Fuzzy E:\Lesson\kitus\lib 1/9 Total: 9

全异步显示文件函数列表，不用的时候不会占用你任何屏幕空间，将 ALT+P 绑定到 :LeaderfFunction! 这个命令上，按 ALT+P 就弹出当前文件的函数列表，然后可以进行模糊匹配搜索，除了上下键移动选择外，各种vim的跳转和搜索命令都可以始用，回车跳转然后关闭函数列表，除此之外按 i 进入模糊匹配，按TAB切换回列表选择。

Leaderf 的函数功能属于你想要它的时候它才会出来，不想要它的时候不会给你捣乱。

文件切换

文件/buffer模糊匹配快速切换的方式，比你打开一个对话框选择文件便捷不少，过去我们常用的 CtrlP 可以光荣下岗了，如今有更多速度更快，匹配更精准以及完美支持后台运行方式的文件模糊匹配工具。我自己用的是上面提到的 [LeaderF](#)，除了提供函数列表外，还支持文件，MRU，Buffer名称搜索，完美代替 CtrlP，使用时需要简单调教下：

```

1 | let g:Lf_ShortcutF = '<c-p>'
2 | let g:Lf_ShortcutB = '<m-n>'
3 | noremap <c-n> :LeaderfMru<cr>
4 | noremap <m-p> :LeaderfFunction!<cr>
5 | noremap <m-n> :LeaderfBuffer<cr>
6 | noremap <m-m> :LeaderfTag<cr>
7 | let g:Lf_StlSeparator = { 'left': '', 'right': '', 'font': '' }
8 |

```

```

9 | let g:Lf_RootMarkers = ['.project', '.root', '.svn', '.git']
10 | let g:Lf_WorkingDirectoryMode = 'Ac'
11 | let g:Lf_WindowHeight = 0.30
12 | let g:Lf_CacheDirectory = expand('~/.vim/cache')
13 | let g:Lf_ShowRelativePath = 0
14 | let g:Lf_HideHelp = 1
15 | let g:Lf_StlColorscheme = 'powerline'
16 | let g:Lf_PreviewResult = {'Function':0, 'BufTag':0}

```

这里定义了 CTRL+P 在当前项目目录打开文件搜索，CTRL+N 打开 MRU搜索，搜索你最近打开的文件，这两项是我用的最频繁的功能。接着 ALT+P 打开函数搜索，ALT+N 打开 Buffer 搜索：

```

37 | \ gui=NONE guifg=DarkGrey guibg=NONE
38
39 | let g:ycm_goto_buffer_command = 'new-or-existing-tab'
40
41
42 | -----
43 | "- FileType Preference
44 | -----
45 | augroup SkywindGroup
46 |   au!
47 |   au FileType python setlocal shiftwidth=4 tabstop=4 noexpandtab omnifunc=pythoncomplete#Complete
48 |   au FileType lisp setlocal ts=8 sts=2 sw=2 et
49 |   au FileType scala setlocal sts=4 sw=4 noet
50 |   au FileType haskell setlocal et
51 | augroup END
52
53
54 | -----
55 | " config
56 | -----
57 | let s:settings = {..
skywind.vim vim utf-8[unix] 19% 47/246 : 99
1 | colors/kolor.vim
2 | asc/keymaps.vim
3 | tools/samples/desktop.ini
4 | skywind.vim
5 | asc/backup.vim
6 | colors/molokai.vim
7 | colors/monokai.vim
8 | colors/molokai2.vim
9 | tools/CMakeLists.txt
LeaderF File NameOnly d:\acm\github\vim 1/15 Total: 214
>>> k!

```

LeaderF 是目前匹配效率最高的，高过 CtrlP/Fzf 不少，敲更少的字母就能把文件找出来，同时搜索很迅速，使用 Python 后台线程进行搜索匹配，还有一个 C 模块可以加速匹配性能，需要手工编译下。LeaderF 在模糊匹配模式下按 TAB 可以切换到匹配结果窗口用光标或者 Vim 搜索命令进一步筛选，这是 CtrlP/Fzf 不具备的，更多方便的功能见它的官方文档。

文件/MRU 模糊匹配对于熟悉的项目效率是最高的，但对于一个新的项目，通常我们都不知道它有些什么文件，那就谈不上根据文件名匹配什么了，我们需要文件浏览功能。如果你喜欢把 Vim 伪装成 NotePad++ 之类的，那你该继续使用 NERDTree 进行文件浏览，但你想按照 Vim 的方式来，推荐阅读这篇文章：

Oil and vinegar – split windows and project drawer

然后像我一样开始使用 vim-dirvish，进行一些配置，比如当前文档按 - 号就能不切窗口的情况下在当前窗口直接返回当前文档所在的目录，再按一次减号就返回上一级目录，按回车进入下一级目录或者再当前窗口打开光标下的文件。进一步映射 <tab>7, <tab>8 和 <tab>9 分别用于在新的 split, vsplit 和新标签打开当前文件所在目录，这样从一个文件入手，很容易找到和该文件相关的其他项目文件。

最后一个是 C/C++ 的头文件/源文件快速切换功能，有现成的插件做这事情，比如 a.vim，我自己没用，因为这事情太简单，再我发现 a.vim 前我就觉得需要这个功能，然后自己两行 vim 脚本就搞定了。

参数提示

这个功能因人而异，有人觉得不需要，有人觉得管用。写 C/C++ 时函数忘了可以用上面的 YCM 补全，但很多时候是参数忘记了怎么办？YCM 的参数提示很蛋疼，要打开个 Preview 窗口，实在是太影响我的视线了，我自己写过一些参数提醒功能，可以在最下面的命令行显示当前函数的参数，不过这是基于 tags 的，搭配前面的 gutentags，对其他语言很管用，但对 C/C++ 我们可以使用 echodoc 插件：


```
217
218 " Restore 'coptions' {{{
219 let &cpo = s:save_cpo
+ 220
Echodoc
```

它可以无缝的和前面的 YCM 搭配，你用 YCM 时 tab 补全了一个函数名后，只要输入左括号，下面命令行就会里面显示出该函数的参数信息，唯一需要设置的是使用 `set noshowmode` 关闭模式提示，就是底部 —INSERT— 那个，我们一般都用 `airline` / `lightline` 之类的显示当前模式了，所以默认模式提示可以关闭，INSERT 模式下的命令行，完全留给 echodoc 显示参数使用。

更多阅读

2018年了，用点新方法，网上那些 Vim 开发 C/C++ 的文章真的都可以淘汰了。

更多参考：《Vim 中文版入门到精通》和《[Vim 中文速查表](#)》。本文主要是针对 C/C++ 环境搭建的插件介绍，关于基本使用，欢迎参考上面这些链接。篇幅有限，这里只能谈一部分内容，对大多数人已经够了，如果你想继续深入的话，还可以慢慢折腾诸如 git 集成，帮助文档集成，调试，rtags 以及 lsp 这些东西。

 [随笔](#)  Vim