YouCompleteMe – 神器在手，代码无忧！ (http://www.yycmmc.com/?p=53)

6-06 3,953 views

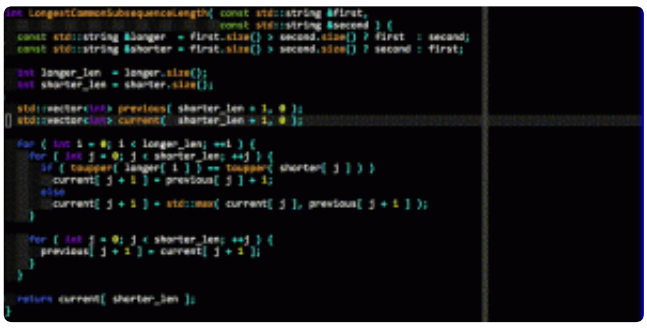
一、YouCompleteMe (https://github.com/Valloric/YouCompleteMe) (YCM) 简介
在整个VIM的发展史中，代码补全一直是补忽略的细节，但却又是开发中不可或缺的一项功能。编辑器与集成开发环境的其中区别就在于多文件之间的浏览和自动补全。长久以来，VIM就定位一实用的编辑工具，而非IDE，这就造成了VIM的先天不足。

“文本编辑器”这种东西一般都不真正的理解程序语言。很多 Emacs 和 vi 的用户以为用 etags 和 ctags 这样的工具就能让他们“跳转到定义”，然而这些 tags 工具其实只是对程序的“文本”做一些愚蠢的正则表达式匹配。它们根本没有对程序进行 parse，所以其实只是在进行一些“瞎猜”。简单的函数定义它们也许能猜对位置，但是对于有重名的定义，或者局部变量的时候，它们就力不从心了。
—-以上摘自《编辑器与IDE》 (http://www.yinwang.org/blog-cn/2013/04/20/editor-ide/)

VIM中类似的补全插件目前有 AutoComplPop, omnicppcomplete, neocomplcache等，但这些插件在仍停留在“瞎猜”级别，并非语义级的补全。在Apple支持的LLVM/clang诞生后，事情迎来了转机。clang拥有强大语义分析能力，为C/C++/Object-C源代码级别的分析和转化提供了可能，基于clang的语意补全插件开始在VIM和emacs上出现。YCM整合实现了多种同类插件的功能，借助于clang工具生成语法树来完成补全，补全精确、速度快、准确性极高而且可以模糊匹配，不管你的C++代码用什么怪异的写法，只要能编译通过，都能补全，即使是C++11的lambda和auto都没有障碍，比codeblock这些根据tag index补全的IDE都要强大。YCM使用方便，不需要特殊的按键，在输入时自动匹配并弹出匹配窗口，使你能够在整个工程里穿梭自如 ^_^ 。

Auto-triggers as you type. 自动触发，键入即提示，无需快捷键（对C函数的补全需要快捷键）
Uses subsequence-based completion filtering.（模糊匹配）
Uses smart heuristics to intelligently rank whatever candidates survive the filtering step.(词频调整的策略)
Offers semantic completions.（语法层面的补全）
Fast, fast, fast!（号称比clang那个插件更快）
Go to Definition(Function，varible)(跳转到函数、变量的定义处)

二、补全效果



(http://www.yycmmc.com/wp-content/uploads/2014/06/ycm-demo.gif)

三、安装
YCM需要最新版本Vim(7.3.584+)的支持，可以选择从代码编译，也可以选择从软件包管理工具中下载。
1. Ubuntu
2. MacOS X
3. Windows

Windows上配置YCM稍显麻烦，不过可以直接下载编译好的，如果不用这个版本的插件而用官方最新版的会匹配不上，功能用不了。
1).下载 VIM 7.4
GVim 7.4 x86 (http://www.vim.org/download.php)
GVim 7.4 x64 (http://lilydjwg.is-programmer.com/pages/19540.html#win-vim)
2) 下载安装VIM插件 Vundle (https://github.com/gmarik/vundle)
3).下载并安装 Windows X86-64 MSI Installer (2.7.7) [1] (https://www.python.org/ftp/python/2.7.7/python-2.7.7.amd64.msi)
确保将python.exe加入环境变量
4).下载 Vim YouCompleteMe for Windows (https://bitbucket.org/Haroogan/vim-youcompleteme-for-windows/)
下载完成后放到vim的插件目录或者vundle目录下
5).下载 llvm-for-windows (https://bitbucket.org/Haroogan/llvm-for-windows/src)
解压后提取里面的 libclang.dll 文件，放到YouCompleteMe目录下的python文件夹中（和ycm_core.pyd文件一起）。
注意：一定要确保平台版本的一致性。
完成上述步骤后，就可以测试一下啦，输入命令

```
:YcmDiags
```

如果没有提示错误，那就说明插件安装好了。如果提示如下错误
[pre]
Runtime Error!
Program: \gvim.exe

R6034
An application has made an attempt to load the C runtime
library incorrectly.
Please contact the application's support team for more
information.
[/pre]

则需要仔细检查PATH环境变量中的每个路径，如果该路径所在目录下有msvcr90.dll，那么就把这个路径从path环境变量中移除，参见 Haroogan (https://bitbucket.org/Haroogan/vim-youcompleteme-for-windows/src) 的文章。
到此，windows下的YCM插件基本安装完成，可以简单的测试下。如果要匹配STL及Platform SDK，则扔需要进一步配置以下文件：

5）配置_vimrc

```
" YouCompleteMe 功能
" 补全功能在注释中同样有效
let g:ycm_complete_in_comments=1
" 允许 vim 加载 .ycm_extra_conf.py 文件，不再提示
let g:ycm_confirm_extra_conf=0
" 开启 YCM 基于标签引擎
let g:ycm_collect_identifiers_from_tags_files=1
" 引入 C++ 标准库tags，这个没有也没关系，只要.ycm_extra_conf.py文件中指定了正确的标准库路径
set tags+=/data/misc/software/misc./vim/stdcpp.tags
" YCM 集成 OmniCppComplete 补全引擎，设置其快捷键
inoremap <leader>; <C-x><C-o>
" 补全内容不以分割子窗口形式出现，只显示补全列表
set completeopt-=preview
" 从第一个键入字符就开始罗列匹配项
let g:ycm_min_num_of_chars_for_completion=1
" 禁止缓存匹配项，每次都重新生成匹配项
let g:ycm_cache_omnifunc=0
" 语法关键字补全
let g:ycm_seed_identifiers_with_syntax=1
" 修改对C函数的补全快捷键，默认是CTRL + space，修改为ALT + ;
let g:ycm_key_invoke_completion = '<M-;>'
" 设置转到定义处的快捷键为ALT + G，这个功能非常赞
nmap <M-g> :YcmCompleter GoToDefinitionElseDeclaration <C-R>=expand("<cword>")<CR><CR>
```

6）配置.ycm_extra_conf.py

YCM对C/C++能提供基于语法的自动补全功能，依靠的是clang的语法解析。按照其官方文档的说明，我们还需要配置一个.ycm_extra_conf.py文件，该文件即可以在vimrc文件中指定，也可以放置在代码工程的根目录。

这个文件其实是一个python脚本，这个文件是每个人的设置都不一样的，当然绝大多数情况（官方上写的是99%）只需要修改 flags 数组中的编译相关参数。里面最重要的参数就是'-I'和'-isystem'下面指定的头文件所在目录。libclang.dll基于语法检查、自动补全/提示的时候就会到这些目录下去找头文件中的函数定义。

以下是官方的一份模板：

```
import os
import ycm_core
flags = [
    '-std=c++11',
    '-stdlib=libc++',
    '-Wno-deprecated-declarations',
    '-Wno-disabled-macro-expansion',
    '-Wno-float-equal',
    '-Wno-c++98-compat',
    '-Wno-c++98-compat-pedantic',
    '-Wno-global-constructors',
    '-Wno-exit-time-destructors',
    '-Wno-missing-prototypes',
    '-Wno-padded',
    '-x',
    'c++',
    '-I',
    '.',
    '-isystem',
    'D:/android-ndk-r9c/platforms/android-19/arch-arm/usr/include/sys',
    '-isystem',
    'D:/android-ndk-r9c/platforms/android-19/arch-arm/usr/include',
    '-isystem',
    'D:/android-ndk-r9c/sources/cxx-stl/llvm-libc++/libcxx/include',
    '-I',
    'D:/work/jni/inc',
    '-I',
    'D:/work/jni/common',
]
compilation_database_folder = ''
if compilation_database_folder:
  database = ycm_core.CompilationDatabase( compilation_database_folder )
else:
  database = None
SOURCE_EXTENSIONS = [ '.cpp', '.cxx', '.cc', '.c', '.m', '.mm' ]
def DirectoryOfThisScript():
  return os.path.dirname( os.path.abspath( __file__ ) )
def MakeRelativePathsInFlagsAbsolute( flags, working_directory ):
  if not working_directory:
    return list( flags )
  new_flags = []
  make_next_absolute = False
  path_flags = [ '-isystem', '-I', '-iquote', '--sysroot=' ]
  for flag in flags:
    new_flag = flag
    if make_next_absolute:
      make_next_absolute = False
      if not flag.startswith( '/' ):
        new_flag = os.path.join( working_directory, flag )
    for path_flag in path_flags:
      if flag == path_flag:
        make_next_absolute = True
        break
      if flag.startswith( path_flag ):
        path = flag[ len( path_flag ): ]
        new_flag = path_flag + os.path.join( working_directory, path )
        break
    if new_flag:
      new_flags.append( new_flag )
```

```
  return new_flags
def IsHeaderFile( filename ):
  extension = os.path.splitext( filename )[ 1 ]
  return extension in [ '.h', '.hxx', '.hpp', '.hh' ]
def GetCompilationInfoForFile( filename ):
  if IsHeaderFile( filename ):
    basename = os.path.splitext( filename )[ 0 ]
    for extension in SOURCE_EXTENSIONS:
      replacement_file = basename + extension
      if os.path.exists( replacement_file ):
        compilation_info = database.GetCompilationInfoForFile(          replacement_file )
        if compilation_info.compiler_flags_:
          return compilation_info
    return None
  return database.GetCompilationInfoForFile( filename )
def FlagsForFile( filename, **kwargs ):
  if database:
    compilation_info = GetCompilationInfoForFile( filename )
    if not compilation_info:
      return None
    final_flags = MakeRelativePathsInFlagsAbsolute(
      compilation_info.compiler_flags_,
      compilation_info.compiler_working_dir_ )
  else:
    relative_to = DirectoryOfThisScript()
    final_flags = MakeRelativePathsInFlagsAbsolute( flags, relative_to )
  return {
    'flags': final_flags,
    'do_cache': True
  }
```

上一篇 (http://www.yycmmc.com/?p=39)

欢迎留言

* 昵称

* 邮箱

网站

赶快发表你的见解吧!

发表评论