
用 **Vim** 搭建开发环境

孔俊

2021-03-15

Contents

1	插件管理	2
2	代码补全	3
3	错误检查	7
4	符号索引	8
5	任务系统	9
6	语法高亮	12
7	文件操作	13
8	调试	17
9	Git	17
10	格式化	17
11	注释	17
12	结语	20

刚接触 Vim 的同学往往因为无法搭建开发环境而“从入门到放弃”，本文旨在帮助这些同学搭建开发环境，聚焦于最核心的开发需求，忽略换配色调字体之类的细枝末节。如果需要开箱即用的 vim 配置（发行版），可以使用 [Spacevim](#)。

本文使用 `neovim-nightly`，但也适用于 Vim 8.2+，不需要读者有任何 VimL 基础，以 C/C++ 为例，但应该适用于任何语言。

1 插件管理

在 Vim 中，插件只是一些脚本，存放在特定的目录中，运行时将它们所在的目录加入到 `runtimepath` 中。Vim 8 内置了插件管理功能，但不支持高级的插件管理功能。`Vimmers` 实现了多个插件管理器，可以自动下载、更新、安装插件，还可以延迟加载、按需加载，提高启动速度。

上古时期流行手动或使用 [Vundle](#) 管理插件，以上两种方式已经落伍了，这里介绍目前比较流行的三个插件管理器：

- [vim-plug](#)：简单易用高效，是目前最流行的插件管理器
- [dein.vim](#)：功能强大，但使用复杂
- [vim-pathogen](#)：另一款流行的插件管理器，没有用过不做评价

以上三款插件管理器风格各不相同，都有大量用户，功能相当完善，根据自己的喜好选取即可。推荐新手选择 `vim-plug`，对启动时间特别敏感的同学可以考虑 `dein.vim`，我的配置在安装 70 余个插件的情况下，启动仅需 60 余秒。

使用 `vim-plug` 安装插件只需要在 `.vimrc` 中写入以下代码：

```
1 call plug#begin('~/.vim/plugged') " 括号里面是插件目录
2                                " 只能在 plug#begin() 和 plug#end()
                                " 之间写安装插件的命令
3 Plug 'junegunn/vim-easy-align'   " 用户名/插件名，默认从 github 下载安
   装
4 Plug 'https://github.com/junegunn/vim-github-dashboard.git' " 从特定
   URL 下载安装
5 call plug#end()
```

使用 `dein.vim` 安装插件：

```
1 set runtimepath+=~/vim/plugged/repos/github.com/Shougo/dein.vim " 将
   dein.vim 添加到 runtimepath
2
                                     " 注意
                                     这是
                                     Ex
                                     命令，
                                     路径
                                     不要
                                     加引号

3 if dein#load_state('~/.vim/plugged') " 参数是插件目录
4     call dein#begin(general#plugin_dir)
5     call dein#add('~/.vim/plugged/repos/github.com/Shougo/dein.vim') "
       安装 dein.vim
6     call dein#add('junegunn/vim-easy-align') " 用户名/插件名，默认从
       github 下载安装
7     call dein#end()
8     call dein#save_state()
9 endif
10
11 if dein#check_install() " 自动安装未安装的插件
12     call dein#install()
13 endif
```

在安装插件的代码后加上这两行设置：

```
1 filetype plugin indent on
2 syntax on
```

这样可以确保特定于文件类型的插件正常工作。

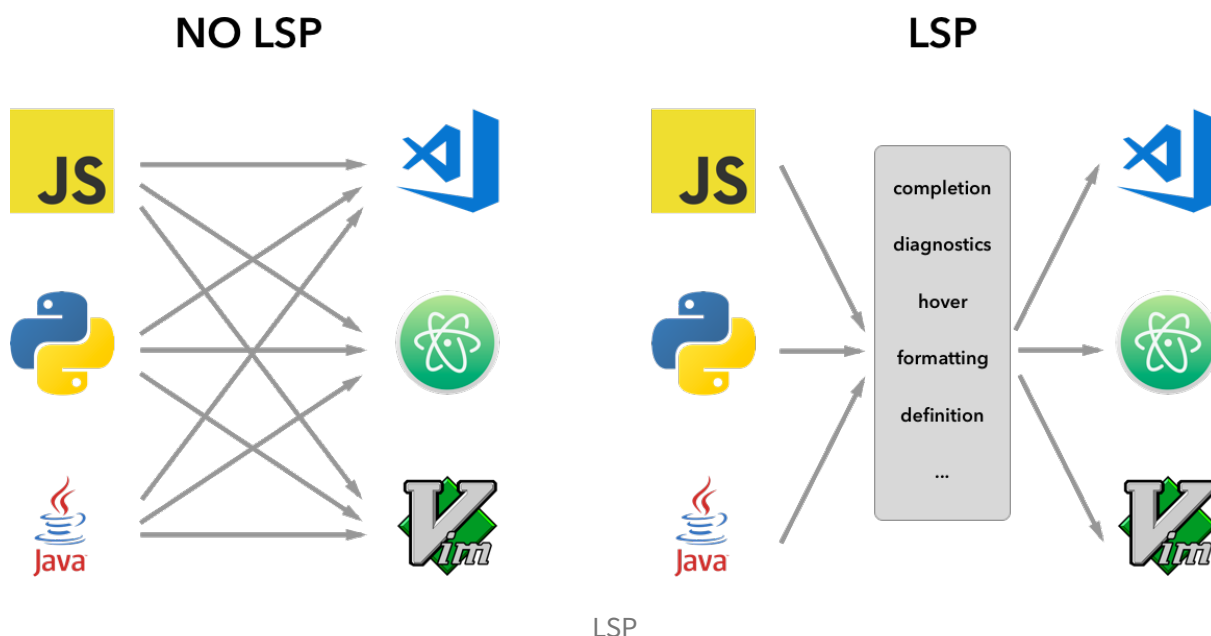
2 代码补全

最简单的代码补全方式是利用 [ctags](#) 生成 **tag** 文件，补全插件解析 **tag** 文件进行补全，这种方式有以下两个好处：

- 最小依赖
- 高效可靠，适用于任何规模的项目

基于 **tag** 的补全不够智能，后来又诞生了 **life-changer** 级别的补全插件 [YouCompleteMe](#)，可以提供 IDE 级别的代码补全。但 [YouCompleteMe](#) 不是开箱即用的，需要下载依赖并编译，并且耦合度比较大，只支持特定语言（主要是 C++）。

目前体验补全体验最好的方式是基于 *LSP* (*Language Server protocol*) 的方案。*LSP* 是一套通信协议，遵从 *LSP* 规范的客户端（各种编辑器/IDE）可以通过众多 *LSP* 服务端按协议标准进行通信，由客户端完成用户界面相关的事情，由服务端提供编程语言相关的：补全，定义引用查找，诊断，帮助文档，重构等服务。架构图如下：



有了 *LSP*，不同的 IDE/编辑器只需要实现 *LSP* 客户端，专心改进用户体验，所有补全的工作都交给 *LSP* 服务器。使用基于 *LSP* 的方案，用户可以在多种语言间无缝切换，让 Vim 支持所有语言（只要有 *LSP* 实现），用户只需要做以下两件事：

- 选择 *LSP* 客户端
- 选择 *LSP* 服务器端

目前 *LSP* 客户端有以下几种选择：

- [vim-lsp](#)
- [LanguageClient-neovim](#)
- neovim-night 内置的 *LSP* 客户端，使用 Lua 语言配置，参考 [nvim-lspconfig](#)
- [coc.nvim](#)
- YouCompleteMe 也提供了对 *LSP* 的支持

coc.nvim 使用 Typescript 开发，是目前最流行、最强大的 *LSP* 客户端，已经发展成了一个 Vim 插件平台，存在大量基于 *coc.nvim* 开发的插件（*coc* 拓展），推荐大家使用 *coc.nvim*。

coc.nvim 依赖于 node.js，但 node.js 似乎已经不再支持 32 位机，因此最新的 *coc.nvim* 很可能无法在 32 位机上运行，请考虑其他几种方案。

```

1  " <Tab> 选择补全候选
2  inoremap <silent><expr> <TAB>
3      \ pumvisible() ? "\<C-n>" :
4      \ <SID>check_back_space() ? "\<TAB>" :
5      \ coc#refresh()
6  inoremap <expr><S-TAB> pumvisible() ? "\<C-p>" : "\<C-h>"
7
8  function! s:check_back_space() abort
9      let col = col('.') - 1
10     return !col || getline('.')[col - 1] =~# '\s'
11 endfunction
12 " gn 跳转到下一个错误, gN 跳转到上一个错误
13 nmap <silent> gn <Plug>(coc-diagnostic-prev)
14 nmap <silent> gN <Plug>(coc-diagnostic-next)
15
16 " gd 跳转到定义, gs 跳转到引用, gt 跳转到类型定义, gK 显示文档
17 nmap <silent> gd <Plug>(coc-definition)
18 nmap <silent> gs <Plug>(coc-references)
19 nmap <silent> gt <Plug>(coc-type-definition)
20 nnoremap <silent> gK :call <SID>show_documentation()<CR>
21 function! s:show_documentation()
22     if (index(['vim','help'], &filetype) >= 0)
23         execute 'h ' . expand('<cword>')
24     elseif (coc#rpc#ready())
25         call CocActionAsync('doHover')
26     else
27         execute '!' . &keywordprg . " " . expand('<cword>')
28     endif
29 endfunction
30
31 " <Leader>f rv 改名, <Leader>f rf 重构
32 nmap <leader>rv <Plug>(coc-rename)
33 nmap <Leader>rf <Plug>(coc-refactor)
34
35 " <C-f> 和 <C-b> 滚动悬浮窗口
36 nnoremap <silent><nowait><expr> <C-f> coc#float#has_scroll() ? coc#
37     float#scroll(1) : "\<C-f>"
38 nnoremap <silent><nowait><expr> <C-b> coc#float#has_scroll() ? coc#
39     float#scroll(0) : "\<C-b>"
40 inoremap <silent><nowait><expr> <C-f> coc#float#has_scroll() ? "\<c-r>=
41     coc#float#scroll(1)\<cr>" : "\<Right>"
42 inoremap <silent><nowait><expr> <C-b> coc#float#has_scroll() ? "\<c-r>=
43     coc#float#scroll(0)\<cr>" : "\<Left>"
44 vnoremap <silent><nowait><expr> <C-f> coc#float#has_scroll() ? coc#
45     float#scroll(1) : "\<C-f>"
46 vnoremap <silent><nowait><expr> <C-b> coc#float#has_scroll() ? coc#
47     float#scroll(0) : "\<C-b>"
48 autocmd User CocJumpPlaceholder call CocActionAsync('showSignatureHelp
49     ')

```

coc.nvim 有自己的配置文件, 叫做 `coc-settings.json`, 一般存放在 `.vim` (neovim 的话在 `~/.config/nvim`)。一般我们会在 `coc-settings.json` 中微调 coc.nvim 和配置 LSP。

目前功能最强的 C++ LSP 服务器是 `ccls`，在 `coc-settings.json` 中配置 `ccls`:

```
1 {
2   "languageserver": {
3     "ccls": {
4       "command": "ccls",
5       "filetypes": ["c", "cc", "cpp", "c++"],
6       "rootPatterns": [".ccls", "compile_commands.json", ".git/",
7         ".root"],
8       "initializationOptions": {
9         "cache": {
10           "directory": ".cache/ccls"
11         },
12       "highlight": {"lsRanges": true }
13     }
14   }
15 }
```

以上配置仅在编辑 C/C++ 文件使用命令 `ccls` 启动 LSP 服务器，将项目根目录设置为包含 `rootPatterns` 中任意文件的目录，索引文件存放在项目根目录的隐藏目录 `.cache/ccls` 中。`highlight` 字段指示 `ccls` 生成高亮信息，基于 LSP 的语法高亮插件会利用 LSP 服务器提供的信息进行精确的语法高亮。

为了让 C++ LSP 服务器知道以程序以何种方法编译，必须要在项目根目录生成 [编译数据库](#) (`compile_commands.json`)。CMake 内置了对编译数据库的支持，只需要在执行 CMake 时加上 `-DCMAKE_EXPORT_COMPILE_COMMANDS=1` 即可; 如果使用 Makefile，可以利用 [Bear](#) 生成编译数据库，通过 `bear make` 来执行 Makefile。Bear 需要执行 Makefile 才能生成编译数据库，如果项目无法正常构建，将不能生成编译数据库，没法使用 LSP 的功能。

有些 vimmer 还基于 `coc.nvim` 开发了一些插件，可以在[这里](#)查看完整的拓展列表。这里给两点建议:

- 尽量不要用 Vim 开发环境开发环境插件管理器安装 `coc.nvim` 拓展

`coc` 基于 `coc.nvim`，使用 Typescript 编写，有些 `coc` 拓展仅支持使用 `coc.nvim` 安装。建议在 `.vimrc` 中定义列表 `let g:coc_global_extensions`，把自己想安装的 `coc` 拓展写进入，`coc.nvim` 会在第一次打开文件时安装。

```
1 let g:coc_global_extensions` = ['coc-vimlsp', 'coc-rust-analyzer']
```

- 优先使用 `coc` 拓展配置 LSP

`coc-rust-analyzer` 之类的 LSP `coc` 拓展通常利用 `coc.nvim` 实现了更多 LSP 功能，请优先使用这些拓展，只有在没有对应语言的 LSP `coc` 拓展时手动配置 LSP。

使用 `ccls`，即使是在 Linux 这种规模的代码仓库中也可以流畅地补全代码。



```
1 file.c +
34 kvfree(fdt->fd);
35 kvfree(fdt->open_fds);
36 kvfree(fdt);
37 }
38
39 static void free_fdt_rcu(struct rcu_head *rcu)
40 {
41     __free_fdt(container_of(rcu, struct fdt, rcu));
42 }
43
44 #define BITBIT_NR(nr)    BITS_TO_LONGS(BITS_TO_LONGS(nr))
45 #define BITBIT_SIZE(nr) (BITBIT_NR(nr) * sizeof(long))
46
47 /*
48  * Copy 'count' fd bits from the old table to the new table and clear the extra
49  * space if any. This does not copy the file pointers. Called with the files
50  * spinlock held for write.
51  */
52 static void copy_fd_bitmaps(struct fdt *nfdt, struct fdt *ofdt,
53                             unsigned int count)
54 {
55     unsigned int cpy, set;
56
57     cpy = count / BITS_PER_BYTE;
58     set = (nfdt->max_fds - count) / BITS_PER_BYTE;
59     memcpy(nfdt->open_fds, ofdt->open_fds, cpy);
60     |
61     memset((char *)nfdt->open_fds + cpy, 0, set);
62     memcpy(nfdt->close_on_exec, ofdt->close_on_exec, cpy);
63     memset((char *)nfdt->close_on_exec + cpy, 0, set);
64
65     cpy = BITBIT_SIZE(count);
66     set = BITBIT_SIZE(nfdt->max_fds) - cpy;
67     memcpy(nfdt->full_fds_bits, ofdt->full_fds_bits, cpy);
68     memset((char *)nfdt->full_fds_bits + cpy, 0, set);
69 }
70
71 /*
72  * Copy all file descriptors from the old table to the new, expanded table and
73  * clear the extra space. Called with the files spinlock held for write.
74  */
75 static void copy_fdt(struct fdt *nfdt, struct fdt *ofdt)
76 {
```

INSERT file.c | + ctags 72% 4% 60:5

code-completion

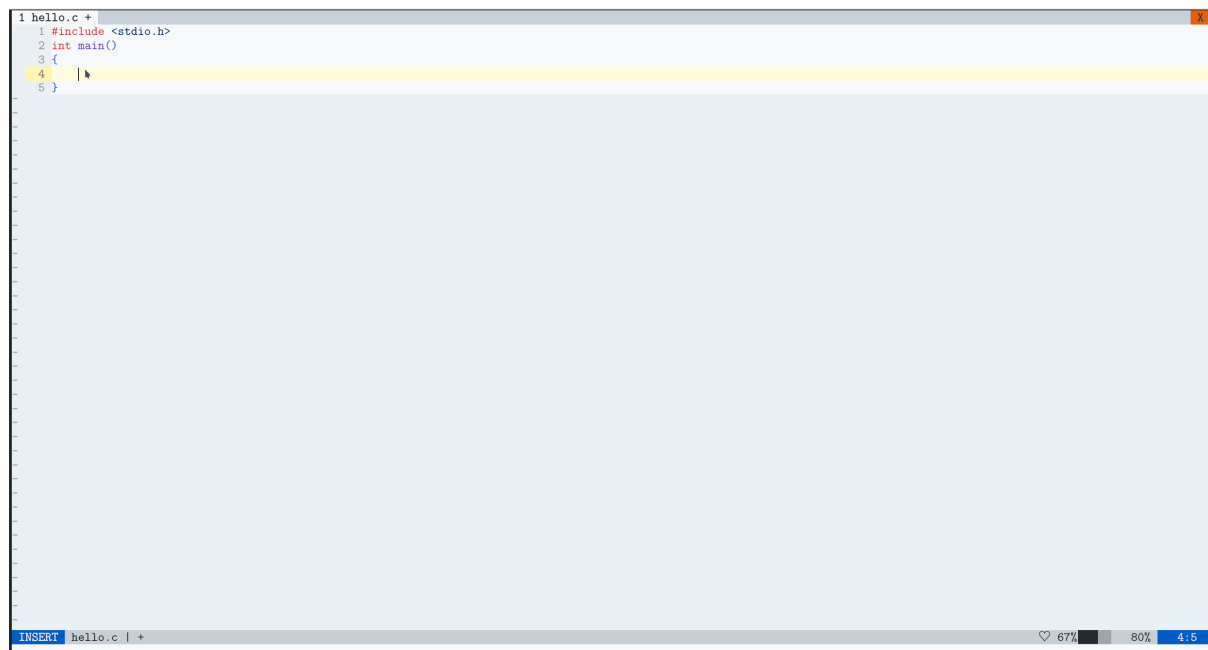
3 错误检查

目前错误检查有两种方案：

- 定时调用外部程序实现实时错误检测
- LSP

如果使用 `coc.nvim`，不需要额外配置，开箱即用。`coc.nvim` 使用 LSP 进行错误检查，不够灵活，无法使用 `linter` 实时检测代码。

基于外部程序的方案非常灵活，比如，可以在基础的错误检测之外同时使用 `clang-tidy` 等工具进行检测。目前这种方案最好的插件是 `ale`，并且 `ale` 可以与 `coc.nvim` 共存，用 `ale` 做实时错误检查，`coc.nvim` 做补全。如果没有特殊需求，直接使用 `coc.nvim` 即可。



dynamic check

4 符号索引

LSP 已经提供了符号索引的功能，可以方便地跳转到定义/引用。通常 LSP 的功能已经够用，但 LSP 存在以下缺点：

- 仅支持单一语言，在多语言项目中无法无法工作

比如可能存在汇编和 C 混合的项目，汇编定义了一个变量在 C 中读写，LSP 无法理解汇编，找不到变量定义的地方。

- LSP 的符号索引功能有限

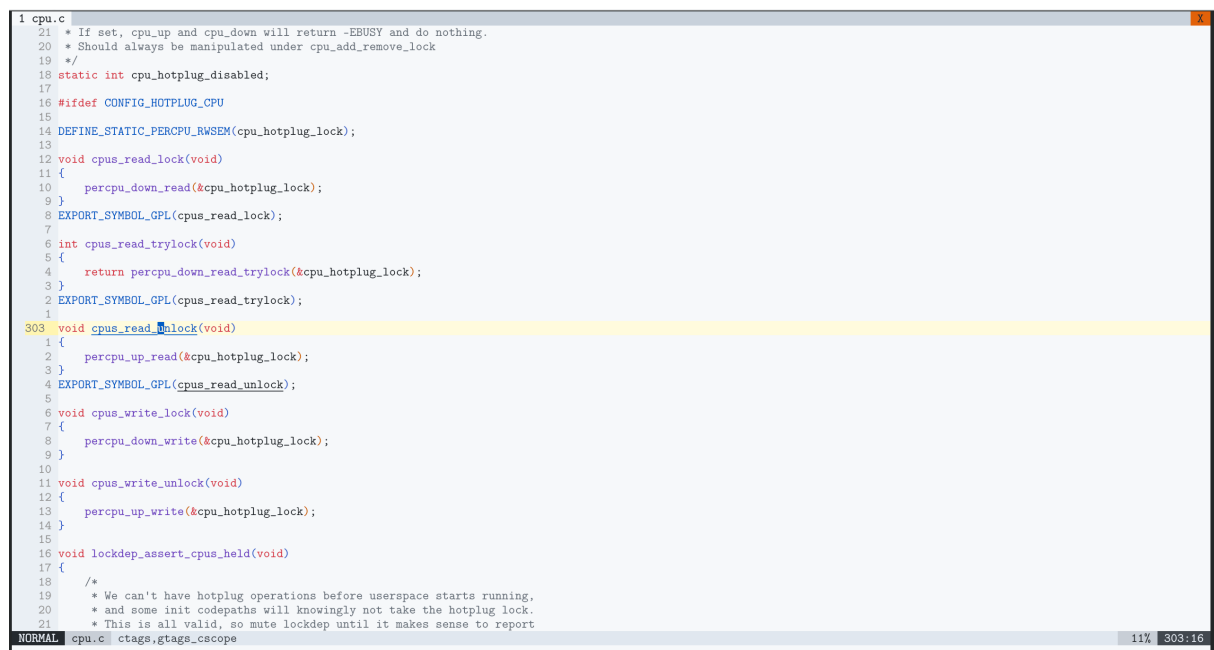
LSP 一般不支持跳转到变量赋值的地方，不支持查找包含该头文件的源文件等。

我们可以使用静态代码索引工具，克服 LSP 的以上缺点。目前静态代码索引最好的方案是 `ctags` 和 `global(gtags)` 混合使用，具体的方法参考韦应笑的深度文章 [Vim 8 中 C/C++ 符号索引：GTags 篇](#)，不再赘述。。

Tips:

- 建议同时使用 LSP 和静态索引工具，LSP 支持的功能用 LSP，不支持的功能或没有 LSP 时用静态索引工具，由于实现这个功能需要用 VimL 编程，这里不介绍，有兴趣的话可以参考我的配置
- `ccls` 实现了更多的功能，如查看类继承体系，查看调用链，查找类的全部实例等。参考配置如下，部分功能用的比较少，就不创建快捷键了，直接使用命令。

```
1 " This comands are defined for ccls(only supports C/C++)
2 command! -nargs=0 Derived :call CocLocations('ccls','$ccls/inheritance
  ', {'derived': v:true})
3 command! -nargs=0 Base :call CocLocations('ccls','$ccls/inheritance')
4 command! -nargs=0 VarAll :call CocLocations('ccls','$ccls/vars')
5 command! -nargs=0 VarLocal :call CocLocations('ccls','$ccls/vars', {'
  kind': 1})
6 command! -nargs=0 VarArg :call CocLocations('ccls','$ccls/vars', {'kind
  ': 4})
7 command! -nargs=0 MemberFunction :call CocLocations('ccls','$ccls/
  member', {'kind': 3})
8 command! -nargs=0 MemberType :call CocLocations('ccls','$ccls/member',
  {'kind': 2})
9 command! -nargs=0 MemberVar :call CocLocations('ccls','$ccls/member',
  {'kind': 4})
10 nmap <silent> gc :call CocLocations('ccls','$ccls/call')<CR>
11 nmap <silent> gC :call CocLocations('ccls','$ccls/call', {'callee': v:
  true})<CR>
```



```
1 cpu.c
21 * If set, cpu_up and cpu_down will return -EBUSY and do nothing.
20 * Should always be manipulated under cpu_add_remove_lock
19 */
18 static int cpu_hotplug_disabled;
17
16 #ifdef CONFIG_HOTPLUG_CPU
15
14 DEFINE_STATIC_PERCPU_RWSEM(cpu_hotplug_lock);
13
12 void cpus_read_lock(void)
11 {
10     percpu_down_read(&cpu_hotplug_lock);
9 }
8 EXPORT_SYMBOL_GPL(cpus_read_lock);
7
6 int cpus_read_trylock(void)
5 {
4     return percpu_down_read_trylock(&cpu_hotplug_lock);
3 }
2 EXPORT_SYMBOL_GPL(cpus_read_trylock);
1
303 void cpus_read_unlock(void)
1 {
2     percpu_up_read(&cpu_hotplug_lock);
3 }
4 EXPORT_SYMBOL_GPL(cpus_read_unlock);
5
6 void cpus_write_lock(void)
7 {
8     percpu_down_write(&cpu_hotplug_lock);
9 }
10
11 void cpus_write_unlock(void)
12 {
13     percpu_up_write(&cpu_hotplug_lock);
14 }
15
16 void lockdep_assert_cpus_held(void)
17 {
18     /*
19      * We can't have hotplug operations before userspace starts running,
20      * and some init codepaths will knowingly not take the hotplug lock.
21      * This is all valid, so mute lockdep until it makes sense to report
```

symbol jump

5 任务系统

在古老的 Vim 工作流中，项目的构建一直是个老大难的问题，要么手动完成，要么自己写简单的脚本完成，VSCode 引入任务系统解决了这个问题，韦易笑大佬的 [asynccrun.vim](#) 和 [asynctasks.vim](#) 又将 VSCode 的任务系统引入到了 Vim 中，彻底改变了 Vim 的工作流。这充分体现了 Vim 的优势，Vim 用

户非常乐于吸收别的编辑器的优点，让 Vim 变得更好。

`asynchrn.vim` 让用户可以异步运行 shell 命令，`asynctasks` 让用户可以将常用的命令写入到配置文件中（`~/vim/tasks.ini` 或项目根目录中的 `tasks.ini`），一次编写多次使用。详细的使用方法请参考插件的中文文档。基本配置如下：

```
1 " 将终端放到 tab 中
2 let g:asynctasks_term_pos = 'tab'
3 " 设置 quickfix 大小
4 let g:asynchrn_open = 10
5 " 设置项目根目录标志
6 " 实际上，许多插件都使用这种方法定位根目录，因此可以定一个变量 g:
   rootmarks,
7 " 将所有插件的根目录标志都设置为 g:rootmarks
8 let g:asynchrn_rootmarks = ['.compile_commands.json', '.ccls', '.git']
```

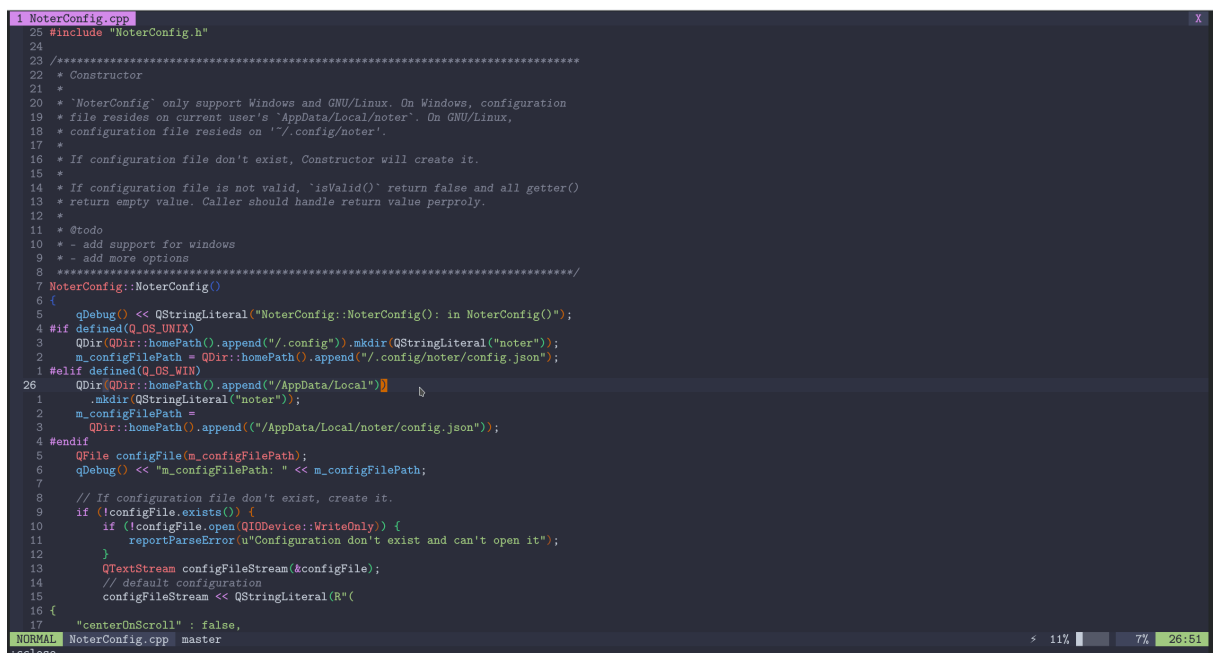
以构建 CMake 项目为例，我需要以不同的模式（Debug/Release）执行 CMake，编译项目，可能还会删除二进制目录，利用这两个 `life-changer` 级别的插件，可以实现一键配置、编译、运行、清理目录。

```
1 [project-build]
2 command = cmake --build _builds -- VERBOSE=1
3 cwd=$(VIM_ROOT)
4 notify=echo
5 save=2
6
7 [project-run]
8 command/linux=_builds/$(VIM_PRONAME)
9 command/win32=_builds\$(VIM_PRONAME).exe
10 cwd=$(VIM_ROOT)
11 output=terminal
12
13 [project-clean]
14 command/linux=rm -rf _builds
15 command/win32=rd/s/q _builds
16 notify=echo
17 cwd=$(VIM_ROOT)
18
19 [project-configure]
20 command/linux=cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=1 -
   DCMAKE_BUILD_TYPE=Debug -S. -B_builds && ln -sf _builds/
   compile_commands.json .
21 command/win32=cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=1 -
   DCMAKE_BUILD_TYPE=Debug -G "MinGW Makefiles" -S. -B_builds && copy
   _builds\compile_commands.json .
22 cwd=$(VIM_ROOT)
23 notify=echo
24 save=2
```

我的 `tasks.ini` 中写好常用的任务命令，您可以直接将我的 `tasks.ini` 直接复制到自己的配置中。

在.vimrc 中映射几个快捷键:

```
1 " 编辑全局 tasks.ini, 随时优化 workflow
2 command! TaskEdit exec 'vsp ' .. general#vimfiles .. '/tasks.ini'
3
4 " 由于 <Tab> 的码值和 <C-I> 相同, 映射这些快捷键后 <C-I> 会变慢
5 noremap <Tab>5 :AsyncTask file-build<cr>
6 noremap <Tab>6 :AsyncTask file-run<cr>
7 noremap <Tab>7 :AsyncTask project-configure<CR>
8 noremap <Tab>8 :AsyncTask project-build<CR>
9 noremap <Tab>9 :AsyncTask project-run<CR>
10 noremap <Tab>0 :AsyncTask project-clean<CR>
```



```
1 NoterConfig.cpp
25 #include "NoterConfig.h"
26
27 /*****
28 * Constructor
29 *
30 * 'NoterConfig' only support Windows and GNU/Linux. On Windows, configuration
31 * file resides on current user's 'AppData/Local/noter'. On GNU/Linux,
32 * configuration file resides on './.config/noter'.
33 *
34 * If configuration file don't exist, Constructor will create it.
35 *
36 * If configuration file is not valid, 'isValid()' return false and all getter()
37 * return empty value. Caller should handle return value properly.
38 *
39 * @todo
40 * - add support for windows
41 * - add more options
42 *****/
43
44 NoterConfig::NoterConfig()
45 {
46     qDebug() << QStringLiteral("NoterConfig::NoterConfig(): in NoterConfig()");
47     #if defined(Q_OS_UNIX)
48     QDir(QDir::homePath().append("/.config")).mkdir(QStringLiteral("noter"));
49     m_configFilePath = QDir::homePath().append("/.config/noter/config.json");
50     #elif defined(Q_OS_WIN)
51     QDir(QDir::homePath().append("/AppData/Local"))
52     .mkdir(QStringLiteral("noter"));
53     m_configFilePath =
54     QDir::homePath().append("/AppData/Local/noter/config.json");
55     #endif
56     QFile configFile(m_configFilePath);
57     qDebug() << "m_configFilePath: " << m_configFilePath;
58     // If configuration file don't exist, create it.
59     if (!configFile.exists()) {
60         if (!configFile.open(QIODevice::WriteOnly)) {
61             reportParseError(u"Configuration don't exist and can't open it");
62         }
63         QTextStream configFileStream(&configFile);
64         // default configuration
65         configFileStream << QStringLiteral(R"(
66 {
67     "centerOnScroll" : false,
68 }
69 )");
70     }
71     NORMAL NoterConfig.cpp master
72 :close
```

build-project

再回头看前面提到的错误检测, 我们可以将执行 linter 的命令写成一个任务, 在代码没有语法错误后调用。

```
1 [clang-tidy]
2 command=find . -type d -name '.cache' -prune -o -type d -name '_builds'
3 -prune -o -name '*.cpp' -print | xargs clang-tidy -checks=level2 -p
4 default/compile_commands.json
5
6 cwd=(VIM_ROOT)
7 notify=echo
```

可能您还想统计本项目的代码量, 也可以通过 asynctask.vim 完成:

```
1 [cloc]
2 command=cloc --exclude-dir=_builds,.* .
3 cwd=(VIM_ROOT)
4 notify=echo
```

只有想象力丰富，`asynrcun.vim` 几乎没有完不成的工作！

您可能会有这样的疑问，假如我定义了 100 个任务，平时只用其中的少数几个任务，岂不是要经常打开 `tasks.ini` 查询？这样的困扰根本不存在，Vim 的另一个强大之处就是插件可以配合工作，我们会在后面介绍解决这个问题的办法。

6 语法高亮

基于正则表达式的语法高亮在 C++ 这种语法非常复杂的语言上表现的很差，2021 年可以彻底抛弃掉这种老掉牙的高亮方案了。请使用 `nvim-treesitter`，它是目前最好的高亮方案（只支持 `neovim-nightly`），如果用 Vim 的话请使用 `vim-lsp-cxx-highlight`。

`vim-lsp-cxx-highlight` 基于 LSP 实现精确的高亮，但存在性能问题，打开文件时有点晃眼，前面 `coc-settings.json` 中已经配置好了 `vim-lsp-cxx-highlight`。

`nvim-treesitter`，基于语义高亮代码，性能强，容错好。



```
12 int tslua_add_language(lua_State *L)
13 {
14     if (lua_gettop(L) < 2 || !lua_isstring(L, 1) || !lua_isstring(L, 2)) {
15         return luaL_error(L, "string expected");
16     }
17     const char *path = lua_tostring(L, 1);
18     const char *lang_name = lua_tostring(L, 2);
19     if (pmmap_has(cstr_t)(langs, lang_name)) {
20         return 0;
21     }
22     140
23     #define BUFSIZE 128
24     char symbol_buf[BUFSIZE];
25     snprintf(symbol_buf, BUFSIZE, "tree_sitter_%s", lang_name);
26     #undef BUFSIZE
27     uv_lib_t lib;
28     if (uv_dlopen(path, &lib)) {
29         snprintf((char *)IObuff, IOSIZE, "Failed to load parser: uv_dlopen: %s",
30                 uv_dLError(&lib));
31         uv_dclose(&lib);
32         lua_pushstring(L, (char *)IObuff);
33         return luaL_error(L);
34     }
35     TSLanguage *(*lang_parser)(void);
36     if (uv_dlsym(&lib, symbol_buf, (void **)&lang_parser)) {
37         snprintf((char *)IObuff, IOSIZE, "Failed to load parser: uv_dlsym: %s",
38                 uv_dLError(&lib));
39         uv_dclose(&lib);
40         lua_pushstring(L, (char *)IObuff);
41         return luaL_error(L);
42     }
43     TSLanguage *lang = lang_parser();
44     if (lang == NULL) {
45         return luaL_error(L, "Failed to load parser: internal error");
46     }
47     pmmap_put(cstr_t)(langs, xstrdup(lang_name), lang);
48     lua_pushboolean(L, true);
49     return 1;
50 }
```

highlight

配置代码如下：

```
1 lua <<EOF
2 require'nvim-treesitter.configs'.setup {
3   ensure_installed = {'c', 'cpp', 'toml', 'json', 'lua', 'python', '
4     bash', 'rust'},
5   highlight = {
6     enable = true,
7   }
8 }
9 -- integrate with rainbow
10 require "nvim-treesitter.highlight"
11 local hlmap = vim.treesitter.highlighter.hl_map
12 hlmap.error = nil
13 hlmap["punctuation.delimiter"] = "Delimiter"
14 hlmap["punctuation.bracket"] = nil
15 EOF
```

Tip: 您可以在 `vimrc` 中进行判断，在 Vim 中使用 `vim-lsp-cxx-highlight`，在 `neovim-nightly` 中使用 `nvim-treesitter`，可以参考我配置中的 `init.vim` 和 `autoload/tools.vim`。

7 文件操作

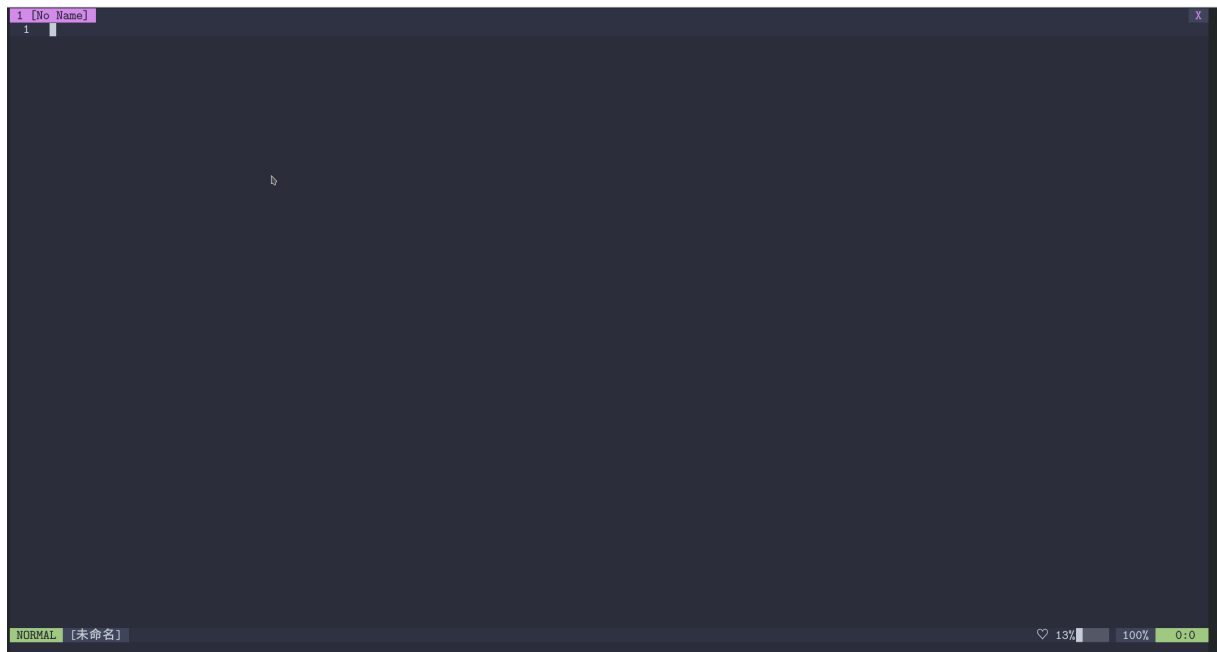
许多 Vim 外的编辑器用户喜欢使用文件树定位项目文件，但 **Vimmer** 更喜欢使用模糊查找插件定位文件。尽管如此，文件树也并非一无用处，在浏览自己不熟悉的项目时，文件树插件可以帮助我们了解项目结构。Vim 自带文件树插件，也有许多 **vimmer** 编写的插件，这里介绍最经典的 [NERDtree](#)。

NERDtree 虽然是最经典的文件树插件，但在许多介绍 Vim 的文章中被骂的狗血临头。许多人批评 **NERDtree** 性能差，在 Linux 这种规模的项目中会直接卡死，但应付中小型项目绰绰有余。

[Leaderf](#) 是国人开发的一款模糊查找插件，性能最强，并且支持许多插件。配置如下：

```
1 let g:Lf_PreviewResult = {
2     \ 'File': 0,
3     \ 'Buffer': 0,
4     \ 'Mru': 0,
5     \ 'Tag': 1,
6     \ 'BufTag': 1,
7     \ 'Function': 1,
8     \ 'Line': 0,
9     \ 'Colorscheme': 0,
10    \ 'Rg': 1,
11    \ 'Gtags': 1
12    \}
13 let g:Lf_PreviewInPopup = 1                " 在 popup 窗口中预览
14     结果
15 let g:Lf_PreviewCode = 1                  " 预览代码
16 let g:Lf_RootMarkers = ['.root', 'compile_command.json', '.git'] "你的
17     根目录标志
18 let g:Lf_WorkingDirectoryMode = 'A'       " 设置 LeaderF 工作目
19     录为项目根目录，如果不在项目中，则为当前目录。
20 let g:Lf_ShortcutF = "<Leader>f"
21 let g:Lf_ShortcutB = "<Leader>bl"
22 noremap <silent><Leader>p :LeaderfFunctionAll<CR> " 搜索函数
23 noremap <silent><Leader>l :LeaderfBufTagAll<CR>   " 搜索缓冲区中的 tag
24 noremap <silent><Leader>d :LeaderfTag<CR>         " 搜索项目中的 tag
25 noremap <silent><Leader>h :LeaderfHelp<CR>       " 搜索 vim help
26 noremap <Leader>rg :Leaderf rg<Space>           " 调用 ripgrep 查找
27     字符串
```

现在，只要按下 <Leader>f，即使是 Linux 这种级别的项目，也能在一瞬间切换到目标文件。



LeaderF

既然 LeaderF 的模糊搜索功能如此强大，能不能让 LeaderF 搜索我们定义的 `asynctask.vim` 任务？答案当然是可以的！


```

1 function! s:lf_task_source(...)
2     let rows = asyncTasks#source(&columns * 48 / 100)
3     let source = []
4     for row in rows
5         let name = row[0]
6         let source += [name . ' ' . row[1] . ' : ' . row[2]]
7     endfor
8     return source
9 endfunction
10
11 function! s:lf_task_accept(line, arg)
12     let pos = stridx(a:line, '<')
13     if pos < 0
14         return
15     endif
16     let name = strpart(a:line, 0, pos)
17     let name = substitute(name, '^s*\(\.{-}\)\s*$', '\1', '')
18     if name != ''
19         exec "AsyncTask " . name
20     endif
21 endfunction
22
23 function! s:lf_task_digest(line, mode)
24     let pos = stridx(a:line, '<')
25     if pos < 0
26         return [a:line, 0]
27     endif
28     let name = strpart(a:line, 0, pos)
29     return [name, 0]
30 endfunction
31
32 function! s:lf_win_init(...)
33     setlocal nonumber
34     setlocal nowrap
35 endfunction
36
37 let g:Lf_Extensions = get(g:, 'Lf_Extensions', {})
38 let g:Lf_Extensions.task = {
39     \ 'source': string(function('s:lf_task_source'))[10:-3],
40     \ 'accept': string(function('s:lf_task_accept'))[10:-3],
41     \ 'get_digest': string(function('s:lf_task_digest'))
42       \ [10:-3],
43     \ 'highlights_def': {
44     \     'Lf_hl_funcScope': '^S\+',
45     \     'Lf_hl_funcDirname': '^S\+\s*\zs<.*>\ze\s*:',
46     \ },
47     \ }
48 noremap <silent><leader>T :Leaderf task<CR> "<leader>T 模糊搜索任务

```

Tips: 使用 [nerdfont](#) 和 [vim-devicons](#) 可以在 LeaderF、NERDtree 等插件中显示漂亮的文件图标。

8 调试

调试一直是 Vim 的弱点，最近 *DAP* (*Debug Adapter Protocol*) 的提出带来了一些改变。*YouCompleteMe* 的主要开发者 *puremourning* 创建了 *vimspector*，这是目前最强的 Vim 调试插件，仍处于开发阶段，您如果有兴趣的话可以参考我的博客 [Vim 最强调试插件：vimspector](#)。

9 Git

Tptope 的 *vim-fugitive* 让 Git 工作流在 Vim 中顺畅无比，使用 *vim-gitgutter* 在侧边栏展示 Git 状态。

```
1  command! -bang -nargs=* -complete=file Make AsyncRun -program=make @ <
    args> " 异步 git push
2  " git-gutter
3  let g:gitgutter_map_keys = 0
4  nmap ghp <Plug>(GitGutterPreviewHunk) " 预览修改 (diff)
5  nmap ghs <Plug>(GitGutterStageHunk)  " 暂存修改
6  nmap ghu <Plug>(GitGutterUndoHunk)    " 丢弃修改
7  nmap [c <Plug>(GitGutterPrevHunk)     " 上一处修改
8  nmap ]c <Plug>(GitGutterNextHunk)     " 下一处修改
```

Tip: *vim-fugitive* 还可以用来处理 *git conflict*，这里不介绍。

10 格式化

注释请使用 *vim-format*，它易于拓展，可以支持所有文件类型。*vim-format* 会根据文件类型执行对应的格式化命令，C/C++ 默认使用 *clang-format*，所以您只需要将 *clang-format* 放到项目根目录即可。

定义一个快捷键快速格式化代码。

```
1  nnoremap <Leader>bf :Autoformat<CR>
```

Tip: 您还可以利用自动命令在写入文件时自动格式化，利用替换命令在写入文件时自动清除行尾空白。

11 注释

目前最流行的注释/反注释是 *nerdcommenter* 和 *vim-commentary*。*nerdcommenter* 相比于 *vim-commentary* 功能更强，拓展性更好，因此推荐使用 *nerdcommenter*。

```
1 " Add spaces after comment delimiters by default
2 let g:NERDSpaceDelims = 1
3
4 " Align line-wise comment delimiters both sides
5 let g:NERDDefaultAlign = 'both'
6
7 " Allow commenting and inverting empty lines (useful when commenting a
  region)
8 let g:NERDCommentEmptyLines = 1
9
10 " Enable trimming of trailing whitespace when uncommenting
11 let g:NERDTrimTrailingWhitespace = 1
12
13 " Enable NERDCommenterToggle to check all selected lines is commented
  or not
14 let g:NERDToggleCheckAllLines = 1
15
16 " Usefull when comment argument
17 let g:NERDAllowAnyVisualDelims = 0
18 let g:NERDAltDelims_asm = 1
```

nerdcommenter 默认的快捷键请参考文档。请不要再蜗牛一样地用 `:help` 命令查看文档，用 `<Leader>h` 模糊搜索！

Tip: 您会发现注释/反注释后光标仍停留在原来的位置，如果您希望光标停留在可视区域结尾，可以添加上以下代码：

```

1 let g:NERDCreateDefaultMappings = 0
2
3 " It is impossible to determine execute mode in hooks. Thus, I wrap raw
  NERDComment()
4 " to pass mode information to hooks and create mappings manually.
5 "
6 " NERDCommenterAltDelims is not wrapped and it would execute hooks. So
  I
7 " delete variable g:NERDCommenter_mode in NERDCommenter_after() to
  disable
8 " hooks executed by NERDCommenterAltDelims
9 function! s:NERDCommenter_wrapper(mode, type) range
10     let g:NERDCommenter_mode = a:mode
11     execute a:firstline .. ',' .. a:lastline 'call NERDComment(' ..
        string(a:mode) .. ',' .. string(a:type) .. ')'
12 endfunction
13
14 " modes: a list of mode(n - normal, x - visual)
15 function! s:create_commenter_mapping(modes, map, type)
16     for l:mode in split(a:modes, '\zs')
17         execute l:mode .. 'noremap <silent> <Leader>' .. a:map .. ' :
            call <SID>NERDCommenter_wrapper(' .. string(l:mode) .. ',' ..
            .. string(a:type) .. ')<CR>'
18     endfor
19 endfunction
20
21 function! CreateCommenterMappings()
22     " All mappings are equal to standard NERDCommenter mappings.
23     call s:create_commenter_mapping('nx', 'cc', 'Comment')
24     call s:create_commenter_mapping('nx', 'cu', 'Uncomment')
25     call s:create_commenter_mapping('n', 'cA', 'Append')
26     call s:create_commenter_mapping('nx', 'c<space>', 'Toggle')
27     call s:create_commenter_mapping('nx', 'cm', 'Minimal')
28     call s:create_commenter_mapping('nx', 'cn', 'Nested')
29     call s:create_commenter_mapping('n', 'c$', 'ToEOL')
30     call s:create_commenter_mapping('nx', 'ci', 'Invert')
31     call s:create_commenter_mapping('nx', 'cs', 'Sexy')
32     call s:create_commenter_mapping('nx', 'cy', 'Yank')
33     call s:create_commenter_mapping('n', 'cA', 'Append')
34     call s:create_commenter_mapping('nx', 'cl', 'AlignLeft')
35     call s:create_commenter_mapping('nx', 'cb', 'AlignBoth')
36     call s:create_commenter_mapping('nx', 'cu', 'Uncomment')
37     call s:create_commenter_mapping('n', 'ca', 'AltDelims')
38     nmap <leader>ca <plug>NERDCommenterAltDelims
39 endfunction
40
41 " NERDCommenter hooks
42 function! NERDCommenter_before()
43     let g:nerdcommenter_visual_flag = v:false
44     if get(g:, 'NERDCommenter_mode', '') =~# '[vsx]' " executed in
        visual mode
45         let l:marklist = getmarklist('%')
46         for l:mark in l:marklist
47             if l:mark['mark'] =~ '>'
48                 let g:nerdcommenter_cursor = l:mark.pos
49                 let g:nerdcommenter_visual_flag = v:true
50                 break
51             endif
52         endfor
53     endif
54 endfunction

```

用 Vim 搭建开发环境

```
1 memory.c
5 /**
6  * @brief 建立物理地址和虚拟地址间的映射
7  *
8  * 本函数仅仅建立映射，不修改物理页引用计数
9  * 当分配物理页失败（创建页表）时 panic,因此不需要检测返回值。
10 */
119
12 * @param page 物理地址
13 * @param addr 虚拟地址
14 * @param flag 标志位
15 * @return 物理地址 page
16 * @see panic(), map_kernel_page()
17 */
18 uint64_t put_page(uint64_t page, uint64_t addr, uint8_t flag)
19 {
20     assert((page & (PAGE_SIZE - 1)) == 0,
21            "put_page(): Try to put unaligned page %p to %p", page, addr);
22     uint64_t vpns[3] = { GET_VPN1(addr), GET_VPN2(addr), GET_VPN3(addr) };
23     uint64_t *page_table = pg_dir;
24     for (size_t level = 0; level < 2; ++level) {
25         uint64_t idx = vpns[level];
26         if (!page_table[idx]) {
27             uint64_t tmp;
28             assert(tmp = get_free_page(),
29                    "put_page(): Memory exhausts");
30             page_table[idx] = (tmp >> 2) | 0x01;
31         }
32         page_table =
33             (uint64_t *)VIRTUAL(GET_PAGE_ADDR page_table[idx]);
34     }
35     page_table[vpns[2]] = (page >> 2) | flag;
36     return page;
37 }
38
39 /**
40  * 将虚拟地址 addr 所在的虚拟页映射到某物理页
41  *
42  * @param addr 虚拟地址
43  * @param flag 标志位
44  * @note 请确保该虚拟地址没有映射到物理页,否则本函数会覆盖原来的映射,导致内存泄漏。
45 */
46 void get_empty_page(uint64_t addr, uint8_t flag)
47 {
48     uint64_t tmp;
49     NORMAL memory.c main
```

nerdcommenter

12 结语

本文介绍了用 Vim 搭建开发环境的思路，但 Vim 的魅力不在于“千篇一律”，而在于“各不相同”，每个 Vim 都有自己的 Vim，根据自己的习惯不断改进工作流。总之，希望本文可以帮助大家走进 Vim 的世界。