Transform Your Neovim into a IDE: A Step-by-Step Guide

MartinLwx included in Vim

2023-02-08 3316 words 16 minutes

CONTENTS

Versions info

I use a Macbook pro-2020 Intel Edition with macOS 13.2. This is my `Nvim` edition:

Code

```
NVIM v0.8.3
Build type: Release
LuaJIT 2.1.0-beta3
Compiled by brew@Ventura

Features: +acl +iconv +tui
See ":help feature-compile"

   system vimrc file: "$VIM/sysinit.vim"
  fall-back for $VIM: "/usr/local/Cellar/neov

Run :checkhealth for more info
```

Why Neovim

After using `Vim` for one year, I find myself having trouble in configure `~/.vimrc`. The syntax of Vimscript is not my liking, leading me to switch `Neovim(Nvim)`. **Rather than** migrating my old `~/.vimrc`. I decided to start from scratch and take this opportunity to re-evaluate my previous Vim configuration. I aim to replace my plugins with the latest SOTA(State-of-the-art) alternatives. ~~It's been some time since I last edited my~~ `~/.vimrc`

file. That's the approach I took in this post. My goal is to make the configuration files self-contained and easily understandable. To achieve this, I aim to provide clear explanations for each setting and include comments to enhance readability.

> 💡 Please note that I may have missed some options. However, as a reminder, you can always access the help docs in the `Nvim` by typing `:h <name>` to get more information

> 💡 This post **assumes** that you have a basic understanding of Vim

The basics

Lua

In my `Nvim` configuration, I will use the Lua programming language as much as possible. Thus, it's recommended that the reader familiarize themselves with Lua. Take a look at Learn Lua in Y minutes

Configurations files paths

The configuration directory for `Nvim` is located at `~/.config/nvim`. On Linux/Mac, `Nvim` will read `~/.config/nvim/init.lua` when it starts up. **Theoretically**, we can put everything inside this single file. It's a bad practice though. To keep things organized, I prefer to break it down into smaller, more manageable parts.

If you follow this post to configure your `Nvim`, your `~/.config/nvim` should look like this⬇️

Code

```
4     ├── colorscheme.lua
5     ├── config
6     │   └── nvim-cmp.lua
7     ├── keymaps.lua
8     ├── lsp.lua
9     ├── options.lua
10    └── plugins.lua
```

The explanations

- `init.lua` is the entry point. **We will "import" other `*.lua` files in `init.lua`**
  - `colorscheme.lua` for the theme
  - `keymaps.lua` for key mappings
  - `lsp.lua` for the LSP support
  - `options.lua` for some global options
  - `plugins.lua` for third-party plugins
- Put the configurations of third-party plugins in this `config` folder. *For example, `nvim-cmp.lua` for the `nvim-cmp` plugin*
- `lua` folder. When we call `require` to import a module in Lua, it will search this folder.
  - **Replace the path separator `/` with `.`, and remove the suffix - `.lua`**. That's how you get the parameter of `require`
  - *For example, to import `nvim-cmp.lua`, you should write `require('config.nvim-cmp')`*

Options

We mainly use these: `vim.g`, `vim.opt`, and `vim.cmd`. I made a cheatsheet below:

In `Vim`	In `Nvim`	Note
`let g:foo = bar`	`vim.g.foo = bar`	

`set foo = bar`	`vim.opt.foo = bar`	`vim.opt.foo = true`
`some_vimscript`	`vim.cmd(some_vimscript)`	

| key mappings

The syntax of key binding in `Nvim` :

Lua

```lua
vim.keymap.set(<mode>, <key>, <action>, <opts
```

For a detailed explanation, please refer to `:h vim.keymap.set`

Configure Nvim from scratch

Now we can configure `Nvim` step by step :)

| Install Neovim

I am a Mac user, so I use Homebrew to install `Nvim` [1]

Code

```
$ brew install neovim
```

After completing the installation, If the `~/.config/nvim/` directory doesn't exist, you should create the folder and `init.lua` file

Code

```
$ mkdir ~/.config/nvim
$ mkdir ~/.config/nvim/lua
$ touch ~/.config/nvim/init.lua
```

> 💡 **Please note that after making any modifications to the `*.lua` files, you need to restart the `Nvim` to see the changes take**

Options configuration

The features:

- Use the system's clipboard
- Use the mouse in `Nvim`
- Tab and whitespace
- UI configuration
- *Smart* search

Create `~/.config/nvim/lua/options.lua` file and edit:

Lua

```lua
-- Hint: use `:h <option>` to figure out the
vim.opt.clipboard = 'unnamedplus'   -- use sy
vim.opt.completeopt = {'menu', 'menuone', 'no
vim.opt.mouse = 'a'                 -- allow

-- Tab
vim.opt.tabstop = 4                 -- number
vim.opt.softtabstop = 4             -- number
vim.opt.shiftwidth = 4              -- insert
vim.opt.expandtab = true            -- tabs a

-- UI config
vim.opt.number = true               -- show a
vim.opt.relativenumber = true       -- add nu
vim.opt.cursorline = true           -- highli
vim.opt.splitbelow = true           -- open n
vim.opt.splitright = true           -- open n
-- vim.opt.termguicolors = true        -- ena
vim.opt.showmode = false            -- we are

-- Searching
vim.opt.incsearch = true            -- search
vim.opt.hlsearch = false            -- do not
vim.opt.ignorecase = true           -- ignore
vim.opt.smartcase = true            -- but ma
```

Then edit the `init.lua` file, use `require` to import `options.lua` file

Key mappings configuration

The features:

- Use `<C-h/j/k/l>` to move the cursor among windows
- Use `Ctrl` + arrow keys to resize windows
- In select mode, we can use `Tab` or `Shift-Tab` to change the indentation repeatedly

Create `~/.config/nvim/lua/keymaps.lua` and edit:

Lua

```
    noremap = true,          non-recursive
    silent = true,         -- do not show messa
}

-----------------
-- Normal mode --
-----------------

-- Hint: see `:h vim.map.set()`
-- Better window navigation
vim.keymap.set('n', '<C-h>', '<C-w>h', opts)
vim.keymap.set('n', '<C-j>', '<C-w>j', opts)
vim.keymap.set('n', '<C-k>', '<C-w>k', opts)
vim.keymap.set('n', '<C-l>', '<C-w>l', opts)

-- Resize with arrows
-- delta: 2 lines
vim.keymap.set('n', '<C-Up>', ':resize -2<CR>
vim.keymap.set('n', '<C-Down>', ':resize +2<C
vim.keymap.set('n', '<C-Left>', ':vertical re
vim.keymap.set('n', '<C-Right>', ':vertical r

-----------------
-- Visual mode --
-----------------

-- Hint: start visual mode with the same area
vim.keymap.set('v', '<', '<gv', opts)
vim.keymap.set('v', '>', '>gv', opts)
```

Edit `init.lua` and import `keymaps.lua`

Lua

```
...
require('keymaps')
```

> 💡 `...` means that we omit other lines(in order to save the length of the post)

Install package manager

A powerful `Nvim` should be augmented with third-party plugins. I have selected Packer.nvim as my

- Support for dependencies
- Expressive configuration and lazy-loading options
- Post-install/update hooks
- ...

> 💡 **The syntax of adding a third-party package is `use ...`**

Create `~/.config/nvim/lua/plugins.lua` and paste the following code. At the moment, I **haven't added any third-party packages**. The template code will do these for us:

1. Install `Packer.nvim` if not installed
2. After modifying the `plugins.lua` file and saving it, Packer.nvim will automatically update and configure the plugins. You should see **a popped window on the right side of the `Nvim`** indicating the status of the plugin updates.

> 💡 You do not need to memorize all the commands available in Packer.nvim, as the template will handle the majority of the work for you. It's worth mentioning that if you failed to update and configure because of the network issue, you can press `<R>` in the popped window to re-sync. Once the Packer.nvim syncs successfully, you can **restart** your `Nvim` to see the changes.

Lua

```lua
local ensure_packer = function()
    local fn = vim.fn
    local install_path = fn.stdpath('data')..
    if fn.empty(fn.glob(install_path)) > 0 th
        fn.system({'git', 'clone', '--depth',
        vim.cmd [[packadd packer.nvim]]
        return true
    end
    return false
end
local packer_bootstrap = ensure_packer()


-- Reload configurations if we modify plugins
-- Hint
--     <afile> - replaced with the filename o
vim.cmd([[
  augroup packer_user_config
    autocmd!
    autocmd BufWritePost plugins.lua source <
  augroup end
]])


-- Install plugins here - `use ...`
-- Packer.nvim hints
--     after = string or list,           -- S
--     config = string or function,      -- S
--     requires = string or list,        -- S
--     ft = string or list,              -- S
--     run = string, function, or table, -- S
return require('packer').startup(function(use
    -- Packer can manage itself
    use 'wbthomason/packer.nvim'

    ----------------------------------------
    -- NOTE: PUT YOUR THIRD PLUGIN HERE --
    ----------------------------------------

    -- Automatically set up your configuratio
    -- Put this at the end after all plugins
    if packer_bootstrap then
        require('packer').sync()
    end
end)
```

```
1 ...
2 require('plugins')
```

If you see a black window with no content when opening `Nvim`, just wait for a moment as Packer.nvim is in the process of installing itself☕

Colorscheme

My favorite theme - monokai. Add this plugin in `plugins.lua`

Lua

```
1 ...
2 use 'tanvirtin/monokai.nvim'
3 ...
```

Save the changes and wait for Packer.nvim to finish installing. Create `~/.config/nvim/colorscheme.lua` and edit:

Lua

```
1 -- define your colorscheme here
2 local colorscheme = 'monokai_pro'
3
4 local is_ok, _ = pcall(vim.cmd, "colorscheme
5 if not is_ok then
6     vim.notify('colorscheme ' .. colorscheme
7     return
8 end
```

The `pcall` here refers to a protected call in Lua, which will return a boolean value to indicate its successful execution(a similar approach can be found in Go with the use of `err`). By using `pcall` instead of `vim.cmd('colorscheme monokai_pro')`, we can avoid some annoying error messages in case the colorscheme is not installed[2]

Again, import `colorscheme.lua` in `init.lua`

Lua

Auto-completion

It can be quite complicated to configure auto-completion manually, which is why we use some fantastic plugins to ease the burden. Now I will discuss **a simpler solution I have found**.

First, use this plugin nvim-cmp, which can manage many completion sources for us. It can also let us customize the completion menu etc.

Create `~/.config/nvim/lua/config/nvim-cmp.lua` and edit

> 💡 Let's first write the configurations of `nvim-cmp` and then modify the `plugins.lua` file. It assures we won't get an annoying error message when the `nvim-cmp` tries to read the missing `nvim-cmp.lua` file. **The code below may seem a little complicated. Don't worry, I will show you how it works.**

Lua

Then we modify `plugins.lua` file to add the plugins needed:

Lua

```
...
use { 'neovim/nvim-lspconfig' }
use { 'hrsh7th/nvim-cmp', config = [[require(
use { 'hrsh7th/cmp-nvim-lsp', after = 'nvim-c
use { 'hrsh7th/cmp-buffer', after = 'nvim-cmp
use { 'hrsh7th/cmp-path', after = 'nvim-cmp'
use { 'hrsh7th/cmp-cmdline', after = 'nvim-cm
use 'L3MON4D3/LuaSnip'
use 'saadparwaiz1/cmp_luasnip'
...
```

Explanations:

find that plenty of plugins follow this API design. **It's a common practice.**

- The `nvim-cmp` is the main plugin we care about. All other plugins begin with `cmp-` is the completion sources helper used by `nvim-cmp`
- `LuaSnip` is a code snippet engine. The `nvim-cmp` says that we should pick a code snippet engine **at least**. Just ignore this if you don't need this
- We can use `config = ...` in Packer.nvim to specify the code to run after the plugin is loaded. *So `config = [[require('config.nvim-cmp')]]` will execute the `nvim-cmp.lua` file. I found this idea on*[3]

Key mappings in nvim-cmp

Use `mapping = ...` . The syntax is `['<key-binding>'] = cmp.mapping.xxx,` . Different `cmp.mapping.xxx` options can be found in the manual. **If you want to set a different key-binding, just change the `[...]`**

My key mappings:

1. Use `<C-k/j>` or "/" to move among completion items
2. Use `<C-b/f>` to scroll among the doc of completion item
3. Use `<CR>` to confirm completion

Completion menu in nvim-cmp

Use `formatting = ...` :

- Use `fields` to specify the appearance of each completion item
- Use `format = function(...)` to set the text for each completion source. You can specify

> 🎙️ You can use basic completion now ~

LSP

To turn `Nvim` into an IDE, it is necessary to rely on LSP[4]. It is cumbersome to install and configure LSP one by one manually, as different LSPs have different installation steps, and it is inconvenient for future management. That's where tools like mason and mason-lspconfig come in to make our lives easier🥰

> ❗ Note that the order of `mason.nvim`, `mason-lspconfig.nvim` and the `nvim-lspconfig` is crucial. **There is a specific ordering requirement between these three plugins and their configurations**. So it's recommended to follow the code provided

Modify the `plugins.lua` file:

Lua

```lua
...
use { 'williamboman/mason.nvim' }
use { 'williamboman/mason-lspconfig.nvim'}
-- use { 'neovim/nvim-lspconfig' }
...
```

Create a `~/.config/nvim/lua/lsp.lua` file to manage it. Let's configure `mason` and `mason-lspconfig` first

Lua

```
4             package_installed = "✓",
5             package_pending = "➜",
6             package_uninstalled = "✗"
7         }
8     }
9 })
10
11 require('mason-lspconfig').setup({
12     -- A list of servers to automatically ins
13     ensure_installed = { 'pylsp', 'gopls', 'l
14 })
```

> 💡 **Add whatever LSP you like in the `ensure_installed`**, the complete list can be found in server_configurations. *I personally use the three programming languages `python/go/rust`, and because we use Lua to configure `Nvim`, we also added `lua_ls` here*

After restarting `Nvim`, you should be able to see in the status bar below that Mason is installing the LSP we specified above (**Note that `Nvim` cannot be closed at this time**). By typing `:Mason` in `Nvim` we can check the installation progress

After successfully installing LSP, we should use `nvim-lspconfig` plug-in to configure (because the configuration code is relatively long, the configuration of `pylsp` is only shown below, and the configuration of other languages is similar). Add the following code to the `nvim/lua/lsp.lua` file

Lua

```
-- LSP list: https://github.com/neovim/nvim-l
-- How to use setup({}): https://github.com/n
--     - the settings table is sent to the LS
--     - on_attach: a lua callback function t
local lspconfig = require('lspconfig')

-- Customized on_attach function
-- See `:help vim.diagnostic.*` for documenta
local opts = { noremap = true, silent = true
vim.keymap.set('n', '<space>e', vim.diagnosti
vim.keymap.set('n', '[d', vim.diagnostic.goto
vim.keymap.set('n', ']d', vim.diagnostic.goto
vim.keymap.set('n', '<space>q', vim.diagnosti

-- Use an on_attach function to only map the
-- after the language server attaches to the
local on_attach = function(client, bufnr)
    -- Enable completion triggered by <c-x><c
    vim.api.nvim_buf_set_option(bufnr, 'omnif

    -- See `:help vim.lsp.*` for documentatio
    local bufopts = { noremap = true, silent
    vim.keymap.set('n', 'gD', vim.lsp.buf.dec
    vim.keymap.set('n', 'gd', vim.lsp.buf.def
    vim.keymap.set('n', 'K', vim.lsp.buf.hove
    vim.keymap.set('n', 'gi', vim.lsp.buf.imp
    vim.keymap.set('n', '<C-k>', vim.lsp.buf.
    vim.keymap.set('n', '<space>wa', vim.lsp.
    vim.keymap.set('n', '<space>wr', vim.lsp.
    vim.keymap.set('n', '<space>wl', function
        print(vim.inspect(vim.lsp.buf.list_wo
    end, bufopts)
    vim.keymap.set('n', '<space>D', vim.lsp.b
    vim.keymap.set('n', '<space>rn', vim.lsp.
    vim.keymap.set('n', '<space>ca', vim.lsp.
    vim.keymap.set('n', 'gr', vim.lsp.buf.ref
    vim.keymap.set('n', '<space>f', function(
end

lspconfig.pylsp.setup({
    on_attach = on_attach,
})
...
```

Append a line in `init.lua`

```
2 require('lsp')
```

The key-binding here is quite similar to what we did in `nvim-cmp`. Refer to the manual as you wish.

Now we got a lightweight IDE🎉🎉🎉

Wrapping up

With this configuration, we successfully turned `Nvim` into a lightweight IDE, which supports code highlighting, code completion, syntax checking, and other functionalities. And it is **completely open source and free**🥰

I realized that even after trying different code editors and IDEs, I always found myself searching for Vim support. So I chose to turn `Nvim` into an IDE, and host the configuration files on my martinlwx/dotfiles. In this way, I can easily clone my configuration files to any new machine and **have a consistent programming experience across machines.**

Polishing tools requires effort and time. In order to understand the purpose of each option, I had to search for various materials. However, despite the challenges, **I firmly believe that it's worth it**. Understanding your tools allows you to further extend and customize them. This article has aimed to present a simple and straightforward configuration, but there are still many beautification and customization things that can be done, including many excellent third-party plug-ins that have not been mentioned yet. The exploration and discovery are left to the readers

Refs

1. Installing-Neovim ↩
2. Adding a colorscheme/theme ↩

Updated on 2023-02-15

Nvim

Back | Home

‹ Type hints: what and why

Linear Regression Model Guide - theory part ›