Extracted from:

Modern Vim

Craft Your Development Environment with Vim 8 and Neovim

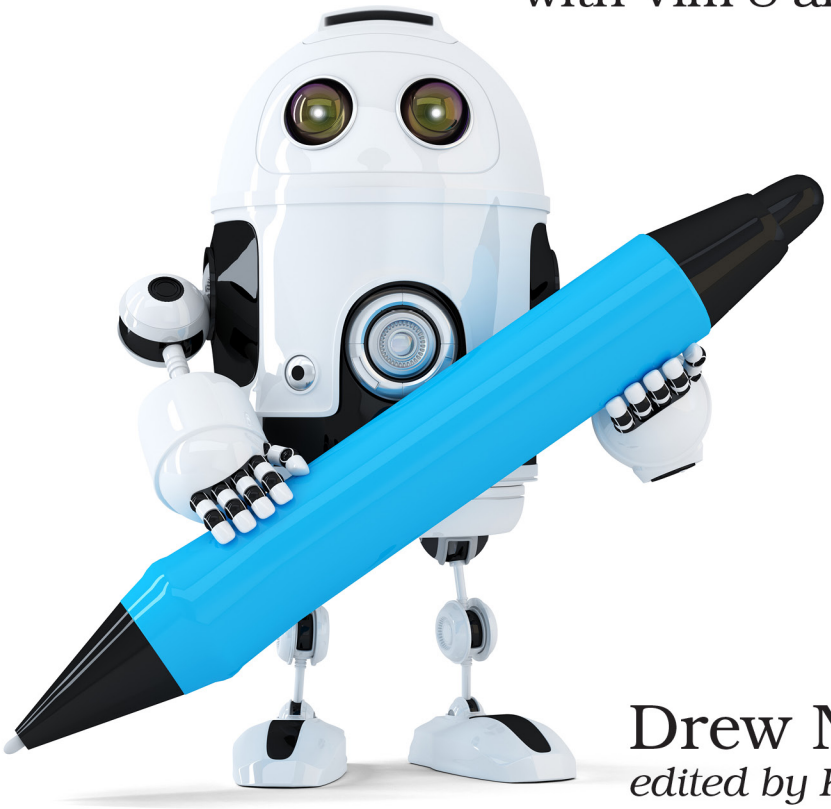
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Modern Vim

Craft Your Development Environment with Vim 8 and Neovim



Drew Neil

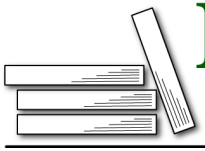
edited by Katharine Dvorak

Modern Vim

Craft Your Development Environment with Vim 8 and Neovim

Drew Neil

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-262-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—November 1, 2017

CHAPTER 5

Neovim's Built-in Terminal Emulator

When you run Neovim in a terminal you're never far from a shell. If you want to run a single command line, you can use `:!{cmd}`. Or if you want to run a series of commands, you can use `<C-z>` to suspend Neovim. This returns you to the shell that you used to launch the `nvim` process. When you're done with the shell, you can bring Neovim back into the foreground using the `fg` command. That brings Neovim back to life exactly as you left it.

You might be wondering: What could I do with Neovim's built-in terminal emulator that I can't already do using this suspend/resume workflow?

Use Normal Mode Commands to Interact with the Shell

When you run a shell inside of Neovim, you can interact with the shell's scrollback using Normal mode commands. That means you can scroll and search using familiar keyboard mappings. You can use text objects to select a range of text. You can yank and paste using any of Neovim's registers. You can jump to a filename under the cursor using the `gf` command.

This is different from enabling vi-mode keybindings in bash, or zsh, or in tmux copy mode. We're not talking about low fidelity vi-emulation. It's what you always wanted: the real thing.

Use Neovim as a Window Manager

With the suspend/resume workflow, you can only have one thing running in the foreground at a time. You're either using Neovim, or you're using the shell.

Suppose you have a `README` file containing instructions on how to get up and running on a new project. You want to be able to keep that file visible so you can follow the instructions, while at the same time executing the specified commands in a shell. You need a window manager.

With Neovim, you could keep the `README` file open in a regular buffer, then open a split containing a shell where you do your work. You can use familiar Neovim commands to create and navigate these windows. (As a bonus, you can copy commands from the `README` file and paste them into the terminal buffer using standard yank/paste commands.)

Control Processes Remotely with Vim Script

When a program is running in a terminal buffer, you can interact with it programatically using Vim script. This is really handy if you want to be able to control that process remotely from Neovim, rather than interacting with the process directly.

Suppose you're doing web development and you need to run a webserver. Sometimes you need to restart the server to make it reload the latest configuration. Usually you achieve this by activating the window where the webserver process is running, stopping the server, then starting it again, before reactivating the window containing Neovim so you can continue with your work. Wouldn't it be handy if you could just run `:Restart` in Neovim?

If you're excited by the possibilities of interacting with running processes programatically, then you're going to love using Neovim's terminal emulator.

Terminal Terminology

When discussing Neovim's terminal emulator, you might be confused by the similar sounding terms: *Terminal mode* and *terminal buffer*. Terminal mode is a mode, just like Normal mode, Insert mode, and so on (we always capitalize mode names in this book, following the convention from Vim's built-in documentation). Just as `nnoremap` lets you create mappings for Normal mode, `tnoremap` lets you create mappings for Terminal mode. You can use Terminal mode only in terminal buffers, where Insert mode is not available.

A regular Vim buffer usually corresponds to a file on disk, whereas a terminal buffer corresponds to a process. You can't directly modify the text in terminal buffer (such as using `dd`). Depending on what program is running within, you may be able to indirectly modify the contents of a terminal buffer by interacting with the underlying program. To interact with the program running inside a terminal buffer, you activate Terminal mode.

As a final note, Vim also has Command-line mode, which shouldn't be confused with Terminal mode. You can activate Command-line mode from Normal mode by pressing `/`, `?` or `:`. This mode allows you to run the search command and to run Ex commands.

Tip 15

Grokking Terminal Mode

Neovim only

As a Vim user, you're used to hopping between modes that are specialized for particular tasks. You spend most of your time in Normal mode, where you can use motions to move around and operators to modify the text in a document. You can switch to Insert mode if you want to add text to a document. Visual mode is useful when you want to select and manipulate text. And Commandline mode lets you run Ex commands such as :w and :q, as well as the search command.

In Neovim, you get a new mode to play with: Terminal mode. In this mode, you can interact with programs that run inside the built-in terminal emulator.

Preparation

In this tip, you'll be running a shell inside of a Neovim terminal buffer. If you use the bash shell with default readline keybindings (also known as emacs-mode), then you should be able to follow this tip seemlessly. If you use a different shell, or if you've customized the keybindings for your shell, then some of the commands may not work for you as described. In this case, you may need to translate the suggested keystrokes to something that works with your setup.

Launching a Shell

If you run the :terminal command with no arguments, Neovim opens a terminal buffer running a shell:

```
⇛ :terminal
```

Having just created a terminal buffer, you start out in Normal mode. Pressing the i key switches you to Terminal mode, which is indicated by the -- TERMINAL -- message at the bottom left of the screen. Pressing the <C-\><C-n> keys switches you back to Normal mode again. This might feel a bit awkward at first. You'll soon find out how to create a mapping to exit Terminal mode more

easily, but for now we'll make do with the defaults. Now try switching between Terminal mode and Normal mode a few times to get used to those commands.

Insert mode is not available in terminal buffers. In regular text buffers, you use `i`, `a`, `I`, and `A` to switch from Normal mode to Insert mode. In terminal buffers, these same keystrokes switch from Normal mode to Terminal mode.

Using Terminal Mode

In Terminal mode, any keys you press will be forwarded to the underlying program (apart from `<C-\><C-n>`, which switches to Normal mode). Right now, the underlying program is a bash shell. Let's switch to Terminal mode and interact with the shell by running some basic commands:

```
⇛ » cd code/terminal
⇛ » pwd
❮ ~/drew/modvim/code/terminal
⇛ » ls
❮ lorem-ipsum.txt                termcursor.vim
  nvim-setup-instructions.md     terminal-mode-escape.vim
  readme.md
⇛ » cat readme.md
❮ Neovim's terminal emulator is cool.
```

In this context, Terminal mode feels similar to Insert mode in that it lets you input text at the current command line. Pressing `<CR>` executes the command line. If you're using the bash shell with default readline bindings (emacs-mode), then you can move the terminal cursor using mappings such as `<C-a>`, `<C-e>`, `<M-b>`, and `<M-f>`. These mappings are interpreted by the underlying program, which in this case is the shell. All that Neovim does is to forward your keystrokes to the program that's running inside the terminal emulator.

Now run the `top` command to launch a new process inside your shell:

```
⇛ » top
```

If you press `?`, you'll see a brief page of documentation for `top`. If you press `q` you'll quit the process and return to your shell. As before, Neovim is simply forwarding your keystrokes to the underlying program, but `top` and `bash` have different ways of interpreting the `q` and `?` keys.

When you're in Terminal mode, your interactions with the underlying program feel just like they would if that program were running in any other terminal emulator. What makes Neovim's terminal emulator special is that fact that you can also switch to Normal mode and use familiar commands to scroll the text, as well as copying and pasting using Vim's registers. We'll explore this

capability in detail in Tip 18, *Using Normal Mode Commands in a Terminal Buffer*, on page ?. First, let's reduce the friction of moving between Normal mode and Terminal mode.

Switching Between Terminal Mode and Normal Mode

When I first started using Neovim's terminal buffers, I kept expecting to be able to use the `<Esc>` key to switch from Terminal mode back to Normal mode. After all, that's how you get back to Normal mode from Insert mode, from Visual mode, and from Commandline mode.

You can use the :tnoremap command to create a mapping that applies only in Terminal mode (:help :tnoremap). Try running this:

```
⇛ :tnoremap <Esc> <C-\><C-n>
```

Now, you can switch from Terminal mode back to Normal mode by pressing they `<Esc>` key. That brings a bit more consistency to the experience of using terminal buffers. But you've created a new problem: You can no longer send an escape key to the program running inside the terminal buffer.

To avoid this problem, create another mapping. Try copying these lines into your vimrc file, then save it and run :source ~/.vimrc:

terminal/terminal-mode-escape.vim

```
if has('nvim')
  tnoremap <Esc> <C-\><C-n>
  tnoremap <C-v><Esc> <Esc>
endif
```

Now you can send an escape key to the terminal by pressing `<C-v><Esc>` (mnemonic: Verbatim escape). I suggest this mapping because it feels idiomatic: In Insert mode, you can use `<C-v>{nondigit}` to enter a nondigit character literally (:help i_ctrl-v). This allows you to insert a tab character by pressing `<C-v><Tab>`, even when the tab key has been configured to insert spaces.

Distinguishing the Terminal Cursor From the Normal Cursor

In a terminal buffer, you have not one but two cursors: the *Terminal cursor*, which is managed by the underlying program, and the *Normal cursor*, which is managed by Vim. This is easier to demonstrate than to describe, so read on for a better understanding.

Type out a short command line, but don't press `<CR>` just yet:

```
⇛ » echo 'hello'
```

While still in Terminal mode, move your cursor to the start of the command line (you can use `<C-a>` if your shell is configured to use emacs bindings). Take note of where your cursor is, then switch to Normal mode. You could go back into Terminal mode either by pressing `i`, `I`, `a`, or `A`. Here's a quiz: Which one would you use if you wanted to switch back to Terminal mode with your cursor placed at the end of the command line?

It's a trick question—the answer is none of them! When you switch to Terminal mode, the cursor always resumes from where it left off. `i`, `I`, `a`, and `A` all do the same thing.

To make this more obvious, try running this command:

```
⇛ :highlight! TermCursorNC guibg=red guifg=white ctermbg=1 ctermfg=15
```

Now switch to Terminal mode and move the cursor at the end of the line (you can use `<C-e>` if your shell is configured with emacs bindings). When you return to Normal mode, the location of the terminal cursor should be picked out in an obvious red.

Try out some Normal mode motions, such as `b`, `w`, `0`, `G`, `k`, and `j`. Vim's cursor moves freely, but the terminal cursor stays put. You can only move the terminal cursor when you're in Terminal mode.

The terminal cursor should be easily visible inside and outside of Terminal mode. If the color scheme you are using doesn't style the TermCursor and TermCursorNC syntax groups (:help hl-TermCursor), I suggest adding these lines to your color scheme:

terminal/termcursor.vim

```
if has('nvim')
  highlight! link TermCursor Cursor
  highlight! TermCursorNC guibg=red guifg=white ctermbg=1 ctermfg=15
endif
```

Of course, you can tweak the colors to match the color scheme's palette.