DB

A Good Vimrc

Posted January 21th, 2014

How To Vimrc

There is just one rule you must follow when crafting your own .vimrc.

> Don't put any lines in your vimrc that you don't understand.

There are tons of tutorials such as this one on the internet that contain all kinds of awesome hacks to make your Vim better, but the absolute worst way to make your environment better is to just copy it wholesale from others.

Spending the time to actually learn what's going into the construction of your editor is invaluable. In the same way that copying notes off a projector by hand often leads to increased information retention, adding features one by one to your vimrc aids in overall Vim comprehension.

With that said, the rest of this article will be me explaining each and every line in my current vimrc in its entirety with the hope that you will find some tricks you haven't seen before. But! My vimrc is far from perfect. I'm always looking for additions that would make my environment better so if you think I missed something important please let me know: @dougblackio.

I will break it up into logical sections.

- Colors

This article will almost certainly fall out of date with my vimrc in the very near future. You can find the most up to date version of it on github.

Colors

```
colorscheme badwolf          " awesome colorscheme
```

Colors! Colorschemes are subjective, but I've currently settled on badwolf by Steve Losh. I found solarized incredibly complete, but a little too bland for my taste. I enjoy colors that pop. I also spend a good deal of time with molokai and still think it's a great scheme, but simply prefer badwolf at the moment.

Moving on:

```
syntax enable           " enable syntax processing
```

The comment should be enough to describe this one. I'll take this moment to plug adding comments to most if not every line in your vimrc. If you're anything like me that file is going to get pretty long, and chances are you won't remember what every line does forever, so adding comments will help Future You know what the hell is going on in there.

Also, many settings in Vim have both a long name and a short name. For instance `background` is the same as `bg`. For future readability, I strongly recommend using the long name.

Spaces & Tabs

The incantations you must throw into your vimrc to get tabs/spaces working the way you want can be pretty confusing, so here's a quick refresher.

```
set tabstop=4       " number of visual spaces per TAB
```

`tabstop` is the number of spaces a tab counts for. So, when Vim opens a file and reads a `<TAB>` character, it uses that many spaces to visually show the `<TAB>`.

```
set softtabstop=4   " number of spaces in tab when editing
```

`softabstop` is the number of spaces a tab counts for *when editing*. So this value is the number of spaces that is *inserted* when you hit `<TAB>` and also the number of spaces that *are removed* when you backspace.

```
set expandtab       " tabs are spaces
```

`expandtab` turns `<TAB>s` into spaces. That's it. So `<TAB>` just becomes a shortcut for "insert four spaces".

Taken together, these are great options for editing files in languages that prefer spaces over tabs, since this ensures no `<TAB>s` are actually used. I spend most of my day in python and bash, where spaces are the norm. I like it, since it means my source code looks the same on every machine.

UI Config

These are options that change random visuals in Vim.

```
set number              " show line numbers
```

Showing line numbers should need no justification.

```
set showcmd             " show command in bottom bar
```

`showcmd` shows the last command entered in the very bottom right of Vim. I have it set here but it's actually not shown in my Vim since I use powerline plugin (which we will get to later).

```
set cursorline          " highlight current line
```

`cursorline` draws a horizontal highlight (or underline, depending on your colorscheme) on the line your cursor is currently on. I've found that this makes it easier to follow exactly what line you left off on when you're switching back to a Vim session or switching between windows in Vim.

```
filetype indent on      " load filetype-specific indent files
```

This both turns on filetype detection and allows loading of language specific indentation files based on that detection. For me, this means the python indentation file that lives at `~/.vim/indent/python.vim` gets loaded every time I open a `*.py` file.

```
set wildmenu            " visual autocomplete for command menu
```

This is a pretty cool feature I didn't know Vim had. You know how Vim automatically autocompletes things like filenames when you, for instance, run `:e ~/.vim<TAB>`? Well it will provide a graphical menu of all the matches you can cycle through if you turn on `wildmenu`.

```
set lazyredraw          " redraw only when we need to.
```

Vim loves to redraw the screen during things it probably doesn't need to—like in the middle of macros. This tells Vim not to bother redrawing during these scenarios, leading to faster macros.

```
set showmatch           " highlight matching [{()}]
```

With `showmatch`, when your cursor moves over a parenthesis-like character, the matching one will be highlighted as well.

Searching

I love Vim's search. I love it even more with the following settings.

```
set incsearch           " search as characters are entered
set hlsearch            " highlight matches
```

These should be pretty self explanatory. They make searching better.

```
" turn off search highlight
nnoremap <leader><space> :nohlsearch<CR>
```

Vim will keep highlighted matches from searches until you either run a new one or manually stop highlighting the old search with `:nohlsearch`. I find myself running this all the time so I've mapped it to `,<space>`.

Folding

Vim folding is a pretty sweet feature that I don't make heavy use of, but when I want it, I want it to have reasonable settings.

```
set foldenable          " enable folding
```

Shows all folds.

```
set foldlevelstart=10   " open most folds by default
```

`foldlevelstart` is the starting fold level for opening a new buffer. If it is set to 0, all folds will be closed. Setting it to 99 would guarantee folds are always open. So, setting it to 10 here ensures that only very nested blocks of code are folded when opening a buffer.

```
set foldnestmax=10      " 10 nested fold max
```

Folds can be nested. Setting a max on the number of folds guards against too many folds. If you need more than 10 fold levels you must be writing some Javascript burning in callback-hell and I feel very bad for you.

```
" space open/closes folds
nnoremap <space> za
```

I change the mapping of `<space>` pretty frequently, but this is its current command. `za` opens/closes the fold around the current block. As an interesting aside, I've heard the `z` character is used to represent folding in Vim because it looks like a folded piece of paper. Probably not, but it makes a nice story. :)

```
set foldmethod=indent   " fold based on indent level
```

This tells Vim to fold based on indentation. This is especially useful for me since I spend my days in Python. Other acceptable values are `marker`, `manual`, `expr`, `syntax`, `diff`. Run `:help foldmethod` to find out what each of those do.

Movement

Here we start getting into custom bindings. This group of bindings all relate to movement commands.

```
" move vertically by visual line
nnoremap j gj
nnoremap k gk
```

These two allow us to move around lines visually. So if there's a very long line that gets visually wrapped to two lines, `j` won't skip over the "fake" part of the visual line in favor of the next "real" line.

```
" move to beginning/end of line
nnoremap B ^
nnoremap E $

" $/^ doesn't do anything
nnoremap $ <nop>
nnoremap ^ <nop>
```

These feel like my most controversial bindings, since they overwrite existing movement bindings. My thinking was that hitting `^` and `$` to jump to the beginning and end of a line was a little too uncomfortable for such an oft-used movement. So I rebound `E` and `B`, which are typically used to move forwards and backwards over visual words to these purposes. Next I bound the old way to `<nop>` to train myself to use the new ones.

```
" highlight last inserted text
nnoremap gV `[v`]
```

This one is pretty cool. It visually selects the block of characters you added last time you were in `INSERT` mode.

Leader Shortcuts

Here we've reached the meat of my custom keybindings. This section will introduce many different plugins and custom functions that I use pretty frequently. Let's get started.

```
let mapleader=","       " leader is comma
```

`\` is a little far away for a leader. I've found `,` to be a much better replacement.

```
" jk is escape
inoremap jk <esc>
```

`<ESC>` is *very* far away. `jk` is a much better replacement as it's on the home row and I actually never type it when writing text. Except right now when I wrote this section of this post. Which I'm writing in Vim. The workaround if you ever need to enter this rare sequence of keys is to enter the `j`, wait for the leader-check timeout to fade, and then enter the `k`.

```
" toggle gundo
nnoremap <leader>u :GundoToggle<CR>
```

In one of its cleverest innovations, Vim doesn't model undo as a simple stack. In Vim it's a tree. This makes sure you never lose an action in Vim, but also makes it much more difficult to traverse around that tree. gundo.vim fixes this by displaying that undo tree in graphical form. Get it and don't look back. Here I've mapped it to `,u`, which I like to think of as "super undo".

```
" edit vimrc/zshrc and load vimrc bindings
nnoremap <leader>ev :vsp $MYVIMRC<CR>
nnoremap <leader>ez :vsp ~/.zshrc<CR>
nnoremap <leader>sv :source $MYVIMRC<CR>
```

These are shortcuts to edit and source my vimrc and my zshrc. That's it.

```
" save session
nnoremap <leader>s :mksession<CR>
```

Ever wanted to save a given assortment of windows so that they're there next time you open up Vim? `:mksession` does just

that! After saving a Vim session, you can reopen it with `vim -S`. Here I've mapped it to `,s`, which I remember by thinking of it as "super save".

```
" open ag.vim
nnoremap <leader>a :Ag
```

The Silver Searcher is a *fantastic* command line tool to search source code in a project. It's wicked fast. The command line tool is named `ag` (like the element silver). Thankfully there is a wonderful Vim plugin ag.vim which lets you use `ag` without leaving Vim *and* pulls the results into a quickfix window for easily jumping to the matches. Here I've mapped it to `,a`.

CtrlP

ctrlp.vim is my life in Vim. If you've never used a fuzzy file searcher this will open your eyes. If you're currently using commandt.vim, you're on the right track, but CtrlP is the spiritual successor. It's *can be* (see below) significantly faster and more configurable than CommandT (Thanks Reddit!). Anyways here are my settings for CtrlP.

```
" CtrlP settings
let g:ctrlp_match_window = 'bottom,order:ttb'
let g:ctrlp_switch_buffer = 0
let g:ctrlp_working_path_mode = 0
let g:ctrlp_user_command = 'ag %s -l --nocolor --hidden -g ""'
```

There are a few things happening here. The first is I'm telling CtrlP to order matching files top to bottom with `ttb`. Next, we tell CtrlP to always open files in new buffers with `let ctrlp_switch_buffer=0`. Setting `let g:ctrlp_working_path=0`

lets us change the working directory during a Vim session and make CtrlP respect that change.

Now, let's talk about speed. CtrlP is entirely written in Vimscript, (which is pretty impressive) but CommandT has parts that are written in C. This means CommandT is, by default, faster than CtrlP. However, we can tell CtrlP to run an external command to find matching files. Now that we have `ag` installed, we can use it with CtrlP to make CtrlP wicked fast.. We do that with the following.

```
let g:ctrlp_user_command = 'ag %s -l --nocolor -g ""'
```

If everything works out, you should see a *noticeable* improvement in the CtrlP speed. There are two caveats to this. Both `g:ctrlp_show_hidden` and `g:ctrlp_custom_ignore` do not work with custom user commands. I only care about the lack of support for custom ignores. Thankfully, `ag` has it's own convention for ignore files: a `.agignore` file that follows the same conventions as `.gitignore`. This is actually great! We only need to define our directories to ignore when searching in one place.

Launch Config

These are options set at launch to configure external tools exactly once.

```
call pathogen#infect()                          " use pathogen
call pathogen#runtime_append_all_bundles()  " use pathogen
```

The `pathogen` options extract all of the Vim plugins from their location in `~/.vim/bundles` to their respective places in the `~/.vim` folder.

Tmux

```
" allows cursor change in tmux mode
if exists('$TMUX')
    let &t_SI = "\<Esc>Ptmux;\<Esc>\<Esc>]50;CursorShape=1\x7\<Esc>\\"
    let &t_EI = "\<Esc>Ptmux;\<Esc>\<Esc>]50;CursorShape=0\x7\<Esc>\\"
else
    let &t_SI = "\<Esc>]50;CursorShape=1\x7"
    let &t_EI = "\<Esc>]50;CursorShape=0\x7"
endif
```

These lines change the cursor from block cursor mode to vertical bar cursor mode when using tmux. Without these lines, tmux always uses block cursor mode.

Autogroups

```
augroup configgroup
    autocmd!
    autocmd VimEnter * highlight clear SignColumn
    autocmd BufWritePre *.php,*.py,*.js,*.txt,*.hs,*.java,*.md
                \:call <SID>StripTrailingWhitespaces()
    autocmd FileType java setlocal noexpandtab
    autocmd FileType java setlocal list
    autocmd FileType java setlocal listchars=tab:+\ ,eol:-
    autocmd FileType java setlocal formatprg=par\ -w80\ -T4
    autocmd FileType php setlocal expandtab
    autocmd FileType php setlocal list
    autocmd FileType php setlocal listchars=tab:+\ ,eol:-
    autocmd FileType php setlocal formatprg=par\ -w80\ -T4
    autocmd FileType ruby setlocal tabstop=2
    autocmd FileType ruby setlocal shiftwidth=2
    autocmd FileType ruby setlocal softtabstop=2
    autocmd FileType ruby setlocal commentstring=#\ %s
    autocmd FileType python setlocal commentstring=#\ %s
    autocmd BufEnter *.cls setlocal filetype=java
    autocmd BufEnter *.zsh-theme setlocal filetype=zsh
    autocmd BufEnter Makefile setlocal noexpandtab
    autocmd BufEnter *.sh setlocal tabstop=2
    autocmd BufEnter *.sh setlocal shiftwidth=2
    autocmd BufEnter *.sh setlocal softtabstop=2
augroup END
```

This is a slew of commands that create language-specific settings for certain filetypes/file extensions. It is important to note they are wrapped in an `augroup` as this ensures the `autocmd`'s are only applied once. In addition, the `autocmd!` directive clears all the `autocmd`'s for the current group.

Backups

If you leave a Vim process open in which you've changed file, Vim creates a "backup" file. Then, when you open the file from a different Vim session, Vim knows to complain at you for trying to edit a file that is already being edited. The "backup" file is created by appending a `~` to the end of the file in the current directory. This can get quite annoying when browsing around a directory, so I applied the following settings to move backups to the `/tmp` folder.

```
set backup
set backupdir=~/.vim-tmp,~/.tmp,~/tmp,/var/tmp,/tmp
set backupskip=/tmp/*,/private/tmp/*
set directory=~/.vim-tmp,~/.tmp,~/tmp,/var/tmp,/tmp
set writebackup
```

`backup` and `writebackup` enable backup support. As annoying as this can be, it is much better than losing tons of work in an edited-but-not-written file.

Custom Functions

I've written a small number of custom functions. Here they are with comments explaining their purpose.

```
" toggle between number and relativenumber
function! ToggleNumber()
    if(&relativenumber == 1)
```

```
        set norelativenumber
        set number
    else
        set relativenumber
    endif
endfunc

" strips trailing whitespace at the end of files. this
" is called on buffer write in the autogroup above.
function! <SID>StripTrailingWhitespaces()
    " save last search & cursor position
    let _s=@/
    let l = line(".")
    let c = col(".")
    %s/\s\+$//e
    let @/=_s
    call cursor(l, c)
endfunction
```

Organization

Once your vimrc starts to fill up, organization becomes an issue. I've grouped this article by logical sections. Not surprisingly, it makes sense to group my actual vimrc by the exact same logical sections. Even cooler, Vim will let us fold all of those sections up by default. So when you open your Vimrc you have a high level view like this:

```
" Doug Black
+--  5 lines: " Colors -----------------------------------
+--  5 lines: " Misc -------------------------------------
+--  9 lines: " Spaces & Tabs ----------------------------
+--  8 lines: " UI Layout --------------------------------
+--  5 lines: " Searching --------------------------------
+--  8 lines: " Folding ----------------------------------
+--  9 lines: " Line Shortcuts ---------------------------
+-- 21 lines: " Leader Shortcuts -------------------------
+--  7 lines: " Powerline --------------------------------
+--  6 lines: " CtrlP ------------------------------------
+--  3 lines: " NERDTree ---------------------------------
+--  4 lines: " Syntastic --------------------------------
+--  6 lines: " Launch Config ----------------------------
+--  9 lines: " Tmux -------------------------------------
```

```
+--  4 lines: " MacVim ----------------------------------
+-- 25 lines: " AutoGroups -----------------------------
+--  7 lines: " Backups ---------------------------------
+-- 50 lines: " Custom Functions ------------------------
```

Here's how we make that happen. First, we tell vim to fold sections by *markers*, rather than *indentation*. That looks like this

```
foldmethod=marker
```

Then we want it to close every fold by default so that we have this high level view when we open our vimrc.

```
foldlevel=0
```

Now, this is a file-specific setting, so we can use a `modeline` to make Vim only use these settings for *this* file. Modelines are special comments somewhere in a file that can can declare certain Vim settings to be used only for that file. So we'll tell Vim to check just the final line of the file for a modeline.

```
set modelines=1
```

Next, we'll add our modeline to the bottom of the file.

```
" vim:foldmethod=marker:foldlevel=0
```

Finally, we need to visually wrap each section in the fold marker. The fold markers are `{{{` and `}}}`.That looks like this.

```
" Section Name {{{
set number "This will be folded
```

```
" }}}
```

That's it. I find this a great way to keep your vimrc highly structured, easy to navigate, and incredibly readable.

Wrapping It Up

I hope this helped you. The reality is that this was a ton of stuff and I still stand by this platitude:

> Don't put anything in your .vimrc you don't understand!

So, if you grab lines from this, make sure you add comments explaining exactly what is going on. If you can't, `:help [setting]` is your best friend.

Thanks for reading! Don't forget to send me your .vimrc tips at @dougblackio.

MORE WORDS