Vim Plugins

Introduction

As we have seen in the *previous chapter*, we can write scripts to extend the existing functionality of Vim to do more stuff. We call these scripts which extend or add functionality as "plugins."

There are various kinds of plugins that can be written:

- vimrc
- global plugin
- filetype plugin
- syntax highlighting plugin
- compiler plugin

Not only can you write your own plugins but also *download and use plugins written by others.*

Customization using vimrc

When I install a new Linux distribution or reinstall Windows, the first thing I do after installing Vim is fetch my latest `vimrc` file from my backups, and then start using Vim. Why is this important? Because the `vimrc` file contains various customizations/settings I like which makes Vim more useful and comfortable for me.

There are two files you can create to customize Vim to your taste:

1. vimrc – for general customizations
2. gvimrc – for GUI specific customizations

These are stored as:

- `%HOME%/_vimrc` and `%HOME%/_gvimrc` on Windows
- `$HOME/.vimrc` and `$HOME/.gvimrc` on Linux/BSD/Mac OS X

See `:help vimrc` on the exact location on your system.

These vimrc and gvimrc files can contain any Vim commands. The convention followed is to use only simple settings in the vimrc files, and advanced stuff are sourced from plugins.

For example, here's a portion of my vimrc file:

```
" My Vimrc file
" Maintainer: www.swaroopch.com/contact/
" Reference: Initially based on
"   http://dev.gentoo.org/~ciaranm/docs/vim-guide/
" License: www.opensource.org/licenses/bsd-license.php

" Enable syntax highlighting.
syntax on

" Automatically indent when adding a curly bracket, etc.
set smartindent

" Tabs should be converted to a group of 4 spaces.
" This is the official Python convention
" (http://www.python.org/dev/peps/pep-0008/)
" I didn't find a good reason to not use it everywhere.
set shiftwidth=4
set tabstop=4
set expandtab
set smarttab

" Minimal number of screen lines to keep above and below the cursor.
set scrolloff=999

" Use UTF-8.
set encoding=utf-8

" Set color scheme that I like.
if has("gui_running")
    colorscheme desert
else
    colorscheme darkblue
endif

" Status line
set laststatus=2
set statusline=
set statusline+=%-3.3n                      " buffer number
set statusline+=%f                          " filename
set statusline+=%h%m%r%w                     " status flags
set statusline+=[%{strlen(&ft)?&ft:'none'}] " file type
set statusline+=%=                           " right align remainder
set statusline+=0x%-8B                       " character value
set statusline+=%-14(%l,%c%V%)               " line, character
set statusline+=%<%P                         " file position
```

```
" Show line number, cursor position.
set ruler

" Display incomplete commands.
set showcmd

" To insert timestamp, press F3.
nmap <F3> a<C-R>=strftime("%Y-%m-%d%a%I:%M%p")<CR><Esc>
imap <F3> <C-R>=strftime("%Y-%m-%d%a%I:%M%p")<CR>

" To save, press ctrl-s.
nmap <c-s> :w<CR>
imap <c-s> <Esc>:w<CR>a

" Search as you type.
set incsearch

" Ignore case when searching.
set ignorecase

" Show autocomplete menus.
set wildmenu

" Show editing mode
set showmode

" Error bells are displayed visually.
set visualbell
```

Notice that these commands are not prefixed by colon. The colon is optional when writing scripts in files because it assumes they are normal mode commands.

If you want to learn detailed usage of each setting mentioned above, refer `:help`.

A portion of my gvimrc file is:

```
" Size of GVim window
set lines=35 columns=99

" Don't display the menu or toolbar. Just the screen.
set guioptions-=m
set guioptions-=T

" Font. Very important.
if has('win32') || has('win64')
    " set guifont=Monaco:h16
```

```
    " http://jeffmilner.com/index.php/2005/07/30/
    \   windows-vista-fonts-now-available/
    set guifont=Consolas:h12:cANSI
elseif has('unix')
    let &guifont="Monospace 10"
endif
```

There are *various examples vimrc files out there* that you should definitely take a look at and learn the various kinds of customizations that can be done, then pick the ones you like and put it in your own vimrc.

A few good ones that I have found in the past are:

It is a known fact that a person's vimrc usually reflects how long that person has been using Vim.

Global plugin

Global plugins can be used to provide global/generic functionality.

Global plugins can be stored in two places:

1. `$VIMRUNTIME/plugin/` for standard plugins supplied by Vim during its installation
2. To install your own plugins or plugins that you have downloaded from somewhere, you can use your own plugin directory:

- `$HOME/.vim/plugin/` on Linux/BSD/Mac OS X
- `%HOME%/vimfiles/plugin/` on Windows
- See `:help runtimepath` for details on your plugin directories.

Let's see how to use a plugin.

A useful plugin to have is `highlight_current_line.vim` by Ansuman Mohanty which does exactly as the name suggests. Download the latest version of the file `highlight_current_line.vim` and put it in your plugin directory (as mentioned above). Now, restart Vim and open any file. Notice how the current line is highlighted compared to the other lines in the file.

But what if you don't like it? Just delete the `highlight_current_line.vim` file and restart Vim.

Similarly, we can install our own `related.vim` or `capitalize.vim` from the *Scripting* chapter into our plugin directory, and this avoids us the trouble of having to use `:source` every time. Ultimately, any kind of Vim plugin that you write should end up somewhere in your `.vim/vimfiles` plugin directory.

Filetype plugin

Filetype plugins are written to work on certain kinds of files. For example, programs written in the C programming language can have their own style of indentation, folding, syntax highlighting and even error display. These plugins are not general purpose, they work for these specific filetypes.

Using a filetype plugin

Let's try a filetype plugin for XML. XML is a declarative language that uses tags to describe the structure of the document itself. For example, if you have a text like this:

```
Iron Gods
---------

Ashok Banker's next book immediately following the Ramayana is said to
be a novel tentatively titled "Iron Gods" scheduled to be published in
2007. A contemporary novel, it is an epic hard science fiction story
about a war between the gods of different faiths. Weary of the
constant infighting between religious sects and their deities, God
(aka Allah, Yahweh, brahman, or whatever one chooses to call the
Supreme Deity) wishes to destroy creation altogether.

A representation of prophets and holy warriors led by Ganesa, the
elephant-headed Hindu deity, randomly picks a sample of mortals, five
of whom are the main protagonists of the book--an American Catholic,
an Indian Hindu, a Pakistani Muslim, a Japanese Buddhist, and a
Japanese Shinto follower.  The mortal sampling, called a 'Palimpsest'
is ferried aboard a vast Dyson's Sphere artifact termed The Jewel,
which is built around the sun itself, contains retransplanted cities
and landscapes brought from multiple parallel Earths and is the size
of 12,000 Earths. It is also a spaceship travelling to the end of
creation, where the Palimpsest is to present itself before God to
plead clemency for all creation.

Meanwhile, it is upto the five protagonists, aided by Ganesa and a few
concerned individuals, including Lucifer Morningstar, Ali Abu Tarab,
King David and his son Solomon, and others, to bring about peace among
the myriad warring faiths.  The question is whether or not they can do
so before the audience with God, and if they can do so peacefully--for
pressure is mounting to wage one final War of Wars to end all war
itself.

(Excerpt taken from
```

```
http://en.wikipedia.org/w/index.php?title=Ashok_Banker&oldid=86219280

under the GNU Free Documentation License)
```

It can be written in XML form (specifically, in 'DocBook XML' format) as:

```
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.5//EN"
        "http://www.oasis-open.org/docbook/xml/4.5/docbookx.dtd">
<article>

    <articleinfo>
        <author><firstname>Wikipedia Contributors</firstname></author>
        <title>Iron Gods</title>
    </articleinfo>

    <para>

        Ashok Banker's next book immediately following the Ramayana is
        said to be a novel tentatively titled "Iron Gods" scheduled to
        be published in 2007. A contemporary novel, it is an epic hard
        science fiction story about a war between the gods of
        different faiths.  Weary of the constant infighting between
        religious sects and their deities, God (aka Allah, Yahweh,
        brahman, or whatever one chooses to call the Supreme Deity)
        wishes to destroy creation altogether.

    </para>

    <para>

        A representation of prophets and holy warriors led by Ganesa,
        the elephant-headed Hindu deity, randomly picks a sample of
        mortals, five of whom are the main protagonists of the
        book--an American Catholic, an Indian Hindu, a Pakistani
        Muslim, a Japanese Buddhist, and a Japanese Shinto follower.
        The mortal sampling, called a 'Palimpsest' is ferried aboard a
        vast Dyson's Sphere artifact termed The Jewel, which is built
        around the sun itself, contains retransplanted cities and
        landscapes brought from multiple parallel Earths and is the
        size of 12,000 Earths.  It is also a spaceship travelling to
        the end of creation, where the Palimpsest is to present itself
        before God to plead clemency for all creation.

    </para>
```

```
    <para>

        Meanwhile, it is upto the five protagonists, aided by Ganesa
        and a few concerned individuals, including Lucifer
        Morningstar, Ali Abu Tarab, King David and his son Solomon,
        and others, to bring about peace among the myriad warring
        faiths. The question is whether or not they can do so before
        the audience with God, and if they can do so peacefully--for
        pressure is mounting to wage one final War of Wars to end all
        war itself.

    </para>

    <sidebar>
        <para>

            (Excerpt taken from
            http://en.wikipedia.org/w/index.php?title=Ashok_Banker&amp;oldid=86219280
            under the GNU Free Documentation License)

        </para>
    </sidebar>

</article>
```

Notice how the structure of the document is more explicit in the XML version. This makes it easier for the tools to convert the XML into other kinds of formats including PDF and print versions. The downside is that writing in XML is more difficult for the human who writes it. So let's see how ftplugins can help a Vim user who writes XML.

1. First, download the *xmledit ftplugin* and put it in your `~/.vim/ftplugin/` directory.
2. Add the following line to your `~/.vimrc`:

   ```
   autocmd BufNewFile,BufRead *.xml source ~/.vim/ftplugin/xml.vim
   ```

(Make sure to specify the correct directory as per your operating system)

This enables the xmledit ftplugin everytime you open a file with extension `.xml`.

3. Open Vim and edit a file called `test.xml`.
4. Type ", and see how the xmledit ftplugin adds the closing tag automatically for you. So, your document should now look like:

5. Now, type another `>` and see how the tag expands so that you can enter more tags. The document should now look like this:

   ```
   <article>

   </article>
   ```

6. You'll notice the cursor is also indented, so that you can write the document in a neatly structured way to reflect the structure of the document.

7. Repeat the process until you have written the full document.

Notice how a special ftplugin for XML makes it much easier for you to write XML documents. This is what ftplugins are usually designed for.

Writing a filetype plugin

Let us try to write our own ftplugin.

In our previous example of using the xmledit.vim ftplugin and writing in the XML format, we saw that we had to write some standard header information at the top for every DocBook XML file (which is the specific format we used). Why not make this automatic in Vim via the use of a ftplugin?

Our xml ftplugin basically needs to add the following information at the top of a 'new' XML file:

```
<?xml version="1.0"?>
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.5//EN"
          "http://www.oasis-open.org/docbook/xml/4.5/docbookx.dtd">
```

So, let's put our new ftplugin, say called 'xmlheader.vim' to work only on 'BufNewFile' event. So, add the following to your `~/.vimrc`:

```
autocmd BufNewFile *.xml source ~/.vim/ftplugin/xmlheader.vim
```

Now, all we need to do in the xmlheader.vim is to set the first and second lines of the file:

```
" Vim plugin to add XML header information to a new XML file
call setline(1, '<?xml version="1.0"?>')
call setline(2, '<!DOCTYPE article
\   PUBLIC "-//OASIS//DTD DocBook XML V4.5//EN"
\   "http://www.oasis-open.org/docbook/xml/4.5/docbookx.dtd">')
```

So, now restart Vim, ensure that 'test.xml' file doesn't already exist, and run `:e test.xml`. You will see that the header is already filled in!

Syntax scheme

We just wrote a DocBook XML file in the previous section. It would have been helpful if there was some color-coding for the XML file for valid DocBook tags to make sure we're writing it correctly. It turns out that this is possible if we just run `:set filetype=docbkxml`. What happens is that Vim uses the syntax file located in `$VIMRUNTIME/syntax/docbkxml.vim`.

The syntax file defines how the parts of the file relate to each other. For example, the syntax file for XML defines what tags are and what colors should be given to tag names, etc.

Using a syntax scheme

Let us see a syntax file in action. Download the *mkd.vim][6] script which is a syntax file for the [Markdown syntax*. Markdown is basically a format in which plain text can be written so that the format can be converted to HTML later.

1. Open a new file in Vim called 'test_markdown.txt'.
2. Run `:set syntax=mkd`
3. Type the following text in the file:

```
# Bengaluru

The name **Bangalore** is an anglicised version of the city's name in
the Kannada language, Bengaluru.

> A popular anecdote (although one contradicted by historical
> evidence) recounts that the 11th-century Hoysala king Veera Ballala
> II, while on a hunting expedition, lost his way in the forest. Tired
> and hungry, he came across a poor old woman who served him boiled
> beans. The grateful king named the place _"benda kaal-ooru"_
> (literally, "town of boiled beans"), which was eventually
> colloquialised to "Bengaluru".

***

(This information has been retrieved from
[Wikipedia](http://en.wikipedia.org/wiki/Bangalore) under the GNU Free
Documentation License.)
```

4. Notice how different parts of the file such as the heading and emphasized words are automatically highlighted. This should hopefully make writing in Markdown syntax easier.

Writing a syntax scheme

Let us now try to write a syntax file of our own for the *AmiFormat* text format.

Syntax highlighting basically revolves around two steps – first is to define the kind of text format we are looking for and the second is to describe how it is to be displayed.

For example, suppose we want to find all instances of `any word` to be displayed in bold. First, we need to match such a pattern in our text and then link the name of this pattern to the kind of display needed:

```
:syntax match ourBold /<b>.*</b>/
:highlight default ourBold term=bold cterm=bold gui=bold
```

The first line says that we are creating a new type of syntax based on matching a pattern, with name 'ourBold' and the regex pattern as specified above.

The second line says that we want to highlight the 'ourBold' syntax. The scheme we are specifying is the default scheme which means it can be overridden by the user or by other color schemes. We can specify three different ways of representing 'ourBold' for the three different kinds of displays that Vim can run in – black and white terminals, color terminals and the GUI (graphical version).

Sometimes we want to specify some task in our text as a todo item and we usually write it in caps 'TODO', but if we want to make it stand out further?

```
:syntax keyword ourTodo TODO FIXME XXX
:hi def link ourTodo Todo
```

First, we define that 'ourTodo' consists of keywords – these are plain simple words that need to be highlighted, and we link this pattern group 'ourTodo' to a pre-existing group called 'Todo' in Vim. There are many such pre-existing groups in Vim which have predefined color schemes. It is best to link our syntax styles to the existing groups. See `:help group-name` for a list of the groups available.

Next, we can have specific blocks of code in the group enclosed in

```
 <tt>..</tt>
```

so how do we highlight this?

```
:syn region amiCode excludenl start=/[code]/ end=/[/code]/
:hi def link amiCode Identifier
```

First, we specify that we are defining a region of text that has a start pattern and end pattern (which is very simple in our case), and then link it to the pre-existing 'Identifier' class.

Similarly, we can proceed to add definitions to other parts of the text as defined in the *AmiFormat reference*, and the final script can look something like this:

```
" Vim syntax file for AmiFormat
" Language: AmiFormat
" Version: 1
" Last Change: 2006-12-28 Thu
" Maintainer: www.swaroopch.com/contact/
" License: www.opensource.org/licenses/bsd-license.php
" Reference: http://orangoo.com/labs/AmiNation/AmiFormat/

"""""""""" Initial Checks """""""""""""""""""""""""""""""""""""""""""""""""

" To be compatible with Vim 5.8. See `:help 44.12`
if version < 600
    syntax clear
elseif exists("b:current_syntax")
    " Quit when a (custom) syntax file was already loaded
    finish
endif

"""""""""" Patterns """""""""""""""""""""""""""""""""""""""""""""""""""""""

" Emphasis
syn match amiItalic /<i>.{-}</i>/
syn match amiBold /<b>.{-}</b>/

" Todo
syn keyword amiTodo TODO FIXME XXX

" Headings
syn match amiHeading /^h[1-6].s+.{-}$/

" Lists
syn match amiList /^s**s+/
syn match amiList /^s*d+.s+/

" Classes
syn match amiClass /^s*%(w+).*%/
syn match amiClass /^s*%{.*}.*%/

" Code
```

```
syn region amiCode excludenl start=/[code]/ end=/[/code]/

" HTML
syn region amiEscape excludenl start=/[escape]/ end=/[/escape]/

" Link
syn match amiLink /".{-}":(.{-})/

" Image
syn match amiImage /!.{-}(.{-})!/

"""""""""" Highlighting """""""""""""""""""""""""""""""""""""""""""""""""""

hi def amiItalic term=italic cterm=italic gui=italic
hi def amiBold term=bold cterm=bold gui=bold

hi def link amiHeading Title
hi def link amiTodo Todo
hi def link amiList PreProc
hi def link amiClass Statement
hi def link amiCode Identifier
hi def link amiEscape Comment
hi def link amiLink String
hi def link amiImage String

"""""""""" Finish """""""""""""""""""""""""""""""""""""""""""""""""""""""""

" Set syntax name
let b:current_syntax = "amifmt"
```

Now that the script actually works, I've uploaded it to the *Vim scripts section* already as I wrote this! Now anyone in the world can use the AmiFormat syntax highlighting in Vim.

To learn more about syntax highlighting scripts in Vim, refer:

- `:help syntax`
- `:help usr_44.txt`
- `:help group-name`
- `:help pattern-overview`
- `:help mysyntaxfile`
- `:help new-filetype`

> Note:
>
> If you want to redraw the screen in case the syntax file is causing the display to be improper, press `CTRL-L`.

Note:

You might have already guessed that when we have previously set the `filetype`, Vim in turn automatically sets the `syntax` to the same name also.

Compiler plugin

Compiler plugins are used for compiling programs written in different languages. It is useful anywhere a transformation is required from a source plain text language to a different format, even if you are writing a plain text file in Markdown want to convert the text to HTML using a transformation program.

Let us explore the use of a compiler plugin for Python.

1. Download the *compiler/python.vim* script and put it in your `~/.vim/compiler/` directory.
2. Put the following line in your `~/.vimrc`:

   autocmd BufNewFile,BufRead *.py compiler python
3. Restart Vim and open a Python file, say `test.py` and enter the following program:

```
#!python
print 'Hello World'
```

4. Run `:make` and you should see successful compilation.
5. Let us intentionally introduce an error in the program by changing the spelling of 'print' to 'pritn':

   pritn 'Hello World'

Now run `:make` and notice that the error is displayed and Vim automatically moves the cursor to the error line!

6. Run `:clist` to see the full list of errors.
7. After fixing an error, you can run `:cnext` to move to the next error.

If you open up the `compiler/python.vim` script that we downloaded, you will notice that it is very simple – there are only two variables defined – one is `makeprg` which defines how to 'make' the file, i.e., how to compile it and the second is `errorformat` which defines the form of the error output of the compiler.

I've written a *compiler plugin for Adobe Flex* using the same two variables.

See `:help write-compiler-plugin` and `:help quickfix` for details on how to write your own compiler plugin.

Homework: Write a global plugin

In order to exercise your newly acquired plugin-writing skills, here's an exercise for you to try out:

> Write a plugin that deletes duplicate lines and deletes redundant blank lines in the document.

You can use either Vim's scripting language or any of the other languages that have an interface with Vim.

If you need "inspiration", see *this Vim Tip*.

How about another one?

> Write a script to fetch the meaning and related words for the current word under the cursor.

Again, if you need "inspiration", see my *lookup.vim plugin*.

Disabling plugins

Suppose you find Vim acting weirdly and suspect a plugin to be the cause, then you can allow Vim to do selective initialization using the `-u` command line argument.

For example, `vim -u NONE` will start up Vim without any initialization scripts being run. This is pure raw Vim running. Use `vim -u your-minimal-initialization.vim` to run only the specific initializations that you need. This option is useful for debugging if any problems you are facing are present in Vim or have been introduced by a plugin, etc.

See `:help -u` and `:help starting` for details.

Summary

We have seen the various kinds of plugins available for Vim, how to use such plugins and how to write such plugins. We now have some idea of how extensible Vim is, and how we can write plugins to make our life easier.
