



下载APP



14 | Vim 脚本简介：开始你的深度定制

2020-08-31 吴咏炜

Vim 实用技巧必知必会

[进入课程 >](#)**讲述：吴咏炜**


时长 21:52 大小 20.04M



你好，我是吴咏炜。

学到今天，我们已经看到了很多的 Vim 脚本，只是还没有正式地把它作为一门语言来介绍。今天，我就正式向你介绍把 Vim 的功能粘合到一起的语言——Vim 脚本（Vim script）。掌握 Vim 脚本的基本语法之后，你就可以得心应手地定制你的 Vim 环境啦。

语法概要

首先，我们需要知道，通过命令行模式执行的命令就是 Vim 脚本。它是一种图灵完全  本语言：图灵完全，说明它的功能够强大，理论上可以完成任何计算任务；脚本语言，说明它不需要编译，可以直接通过解释方式来执行。

当然，这并没有说出 Vim 脚本的真正特点。下面，我们就通过各个不同的角度，进行了解，把 Vim 脚本这头“大象”的基本形状完整地摸出来。


在这一讲里，我们改变一下惯例，除非明确说“正常模式命令”，否则用代码方式显示的都是脚本文件里的代码或者命令行模式命令，也就是说，它们前面都不会加 `:`。毕竟我们这一讲介绍的全是 Vim 脚本，而不是正常模式的快捷操作。

打印输出和字符串

学习任何一门语言，我们常常以“Hello world!”开始。对于 Vim 脚本，我们不妨也这样——毕竟，打印是一种重要的调试方式，尤其对于没有专门调试器的脚本语言来说。

Vim 脚本的“Hello world!”是下面这样的：

```
1 echo 'Hello world!'
```

 复制代码

`echo` 是 Vim 用来显示信息的内置命令，而 `'Hello world!'` 是一个字符串字面量。Vim 里也可以使用 `"` 来引起一个字符串。`'` 和 `"` 的区别和在 shell 里比较相似，前者里面不允许有任何转义字符，而后者则可以使用常见的转义字符序列，如 `\n` 和 `\u....` 等。和 shell 不同的是，我们可以在 `'` 括起的字符里把 `'` 重复一次来得到这个字符本身，即 `'It''s'` 相当于 `"It's"`。不过，在这个例子里，显然还是后者更清晰了。

因为 `"` 还有开始注释的作用，一般情况下我推荐在 Vim 脚本里使用 `'`，除非你需要转义字符序列或者需要把 `'` 本身放到字符串里。

字符串可以用 `.` 运算符来拼接。由于字典访问也可以用 `.`，为了避免歧义，Bram 推荐开发者在新的 Vim 脚本中使用 `..` 来拼接。但要注意，这个写法在 Vim 7 及之前的版本里不支持。我目前仍暂时使用 `.` 进行字符串拼接，并和其他大部分运算符一样，前后空一格。这样跟不空格的字典用法比起来，差异就相当明显了。

除了 `echo`，Vim 还可以用 `echomsg`（缩写 `echom`）命令，来显示一条消息。跟 `echo` 不同的是，这条消息不仅会显示在屏幕上，还会保留在消息历史里，可以在之后用 `message` 命令查看。

变量

跟大部分语言一样，Vim 脚本里有变量。变量可以用 `let` 命令来赋值，如下所示：

```
1 let answer = 42
```

[复制代码](#)

然后你当然就可以使用 `answer` 这个变量了，如：

```
1 echo 'The meaning of life, the universe and everything is ' . answer
```

[复制代码](#)

Vim 的变量可以手工取消，需要的命令是 `unlet`。在你写了 `unlet answer` 之后，你就不能再读取 `answer` 这个变量了。

数字

上面的赋值语句用到了整数。Vim 脚本里的数字支持整数和浮点数，在大部分平台上，两者都是 64 位的有符号数字类型，基本对应于大部分 C 语言环境里的 `int64_t` 和 `double`。表示方式也和 C 里面差不多：整数可以使用 `0`（八进制）、`0b`（二进制）和 `0x`（十六进制）前缀；浮点数含小数点（不可省略），可选使用科学计数法。

复杂数据结构


Vim 脚本内置支持的复杂数据结构是列表（list）和字典（dictionary）。这两者都和 Python 里的对应数据结构一样。对于 C++ 的程序员来说，列表基本上就是数组 /array/vector，但大小可变，而且可以直接使用方括号表达式来初始化，如：

```
1 let primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

[复制代码](#)

然后你可以用下标访问，比如用 `primes[0]` 就可以得到 2。

字典基本上就是 map，可以使用花括号来初始化，如：

 复制代码

```
1 let int_squares = {  
2     \0: 0,  
3     \1: 1,  
4     \2: 4,  
5     \3: 9,  
6     \4: 16,  
7     \}
```

键会自动转换成字符串，而值会保留其类型。上面也用到了 Vim 脚本的续行——下一行的第一个非空白字符如果是 \，则表示这一行跟上一行在逻辑上是同一行，这一点和大部分其他语言是不同的。

访问字典里的某一个元素可以用方括号（跟大部分语言一样），如 `int_squares['2']`；或使用 `.`，如 `int_squares.2`。

表达式


跟大部分编程语言类似，Vim 脚本的表达式里可以使用括号，可以调用函数（形如 `func(...)`），支持加（+）、减（-）、乘（*）、除（/）和取模（%），支持逻辑操作（&&、|| 和 !），还支持三元条件表达式（`a ? b : c`）。前面我们已经学过，可以使用 `[]` 访问列表成员，可以使用 `[]` 或 `.` 访问字典的成员，也可以使用 `.` 或 `..` 进行字符串拼接。`==` 和 `!=` 运算符对所有类型都有效，而 `<`、`>=` 等运算符对整数、浮点数和字符串都有效。

对于文本处理，常见的情况是我们使用 `=~` 和 `!~` 进行正则表达式匹配，前者表示匹配的判断，后者表示不匹配的判断。比较操作符可以后面添加 `#` 或 `?` 来强制进行大小写敏感或不敏感的匹配（缺省受 Vim 选项 `ignorecase` 影响）。表达式的左侧是待匹配的字符串，右侧则是用来匹配的正则表达式。

注意表达式不是一个合法的 Vim 命令或脚本语句。在表达式的左侧，需要有 `echo` 这样的命令。如果你只想调用一个函数，而不需要使用其返回的结果，则应使用 `call func(...)` 这样的写法。

此外，我们在插入模式和命令行模式下都可以使用按键 `<C-R>=`（两个键）后面跟一个表达式来使用表达式的结果。在替换命令中，我们在 `\=` 后面也同样可以跟一个表达式，来表

示使用该表达式的结果。比如，下面的命令可以在当前编辑文件的每一行前面插入行号和空格：


 复制代码

```
1 :%s/^/\=line('.') . ' ' /
```

line 是 Vim 的一个内置函数，line('.') 表示“当前”行的行号，剩下部分你应该直接就明白了吧？

控制结构

作为一门完整的编程语言，标准的控制结构当然也少不了。Vim 支持标准的 if、while 和 for 语句。语法上，Vim 的写法有点老派，跟当前的主流语言不太一样，每种结构都要用一个对应的 endif、endwhile 和 endfor 来结束，如下面所示：

 复制代码

```
1 " 简单条件语句
2 if 表达式
3     语句
4 endif
5
6 " 有 else 分支的条件语句
7 if 表达式
8     语句
9 else
10    语句
11 endif
12
13 " 更复杂的条件语句
14 if 表达式
15     语句
16 elseif 表达式
17     语句
18 else
19     语句
20 endif
21
22 " 循环语句
23 while 表达式
24     语句
25 endwhile
```

在 `while` 和 `for` 循环语句里，你可以使用 `break` 来退出循环，也可以使用 `continue` 来跳过循环体内的其他语句。作为一个程序员，理解它们肯定没有任何困难。

Vim 脚本的 `for` 语句跟 Python 非常相似，形式是：

```
1 for var in object
2     这儿可以使用 var
3 endfor
```

[复制代码](#)

表示遍历 `object`（通常是个列表）对象里面的所有元素。

哦，跟 Python 一样，Vim 脚本也没有 `switch/case` 语句。

函数和匿名函数

为了方便开发，函数肯定也是少不了的。Vim 脚本里定义函数使用下面的语法：

```
1 function 函数名(参数1, 参数2, ...)
2     函数内容
3 endfunction
```

[复制代码](#)

Vim 里用户自定义函数必须首字母大写（和内置函数相区别），或者使用 `s:` 表示该函数只在当前脚本文件有效。... 可以出现在参数列表的结尾，表示可以传递额外的无名参数。使用有名字的参数时，你需要加上 `a:` 前缀。要访问额外参数，则需要使用 `a:1`、`a:2` 这样的形式。特殊名字 `a:0` 表示额外参数的数量，`a:000` 表示把额外参数当成列表来使用，因而 `a:000[0]` 就相当于 `a:1`。

在函数里面，跟大部分语言一样，你可以使用 `return` 命令返回一个结果，或提前结束函数的执行。

Vim 脚本里允许匿名函数，形式是 {逗号分隔开的参数 -> 表达式}。如果你对函数式编程完全没有概念，你可以跳过匿名函数。如果你喜欢函数式编程，那你应该会很欣喜地看到，在 Vim 脚本里可以使用类似下面的语句：


```
1 echo map(range(1, 5), {idx, val -> val * val})
```

结果是 [1, 4, 9, 16, 25]。跟常见的 map 函数不同，Vim 会传过去两个参数，分别是列表索引和值；同时，它会修改列表的内容。不想修改的话，要把列表复制一份，如 `copy(mylist)`。

Vim 特性

上面描述的只是一般性的编程语言语法，但 Vim 脚本如果只当作通用编程语言来用的话，就没啥意义了。我们使用 Vim 脚本，肯定是为了和 Vim 进行交互。下面我们就来仔细检查一下 Vim 脚本里的 Vim 特性。

变量的前缀

我们上面已经提到了变量的 a：前缀。变量的前缀实际上有更多，通用编程概念上很容易理解的是下面四个：

a：表示这个变量是函数参数，只能在函数内使用。

g：表示这个变量是全局变量，可以在任何地方访问。

l：表示这个变量是本地变量，但一般这个前缀不需要使用，除非你跟系统的某个名字发生了冲突。

s：表示这个变量（或函数，它也能用在函数上）只能用于当前脚本，有点像 C 里面的 static 变量和函数，只在当前脚本文件有效，因而不会影响到其他脚本文件里定义的有冲突的名字。

一般编程语言里没有的，是下面这些前缀：

b：表示这个变量是当前缓冲区的，不同的缓冲区可以有同名的 b：变量。比如，在 Vim 里，`b:current_syntax` 这个变量表示当前缓冲区使用的语法名字。

w：表示这个变量是当前窗口的，不同的窗口可以有同名的 w：变量。

t：表示这个变量是当前标签页的，不同的标签页可以有同名的 t：变量。

`v`: 表示这个变量是特殊的 Vim 内置变量, 如 `v:version` 是 Vim 的版本号, 等等 (详见 [☞:help v:var](#))。

还有下面这些前缀, 可以让我们像使用变量一样使用环境变量和 Vim 选项:

`$` 表示紧接着的名字是一个环境变量。注意, 一些环境变量是由 Vim 自己设置的, 如 `$VIMRUNTIME`。

`&` 表示紧接着的名字是一个选项, 比如, `echo &filetype` 和 `set filetype?` 效果相似, 都能用来显示当前缓冲区的文件类型。

`&g`: 表示访问一个选项的全局 (global) 值。对于有本地值的选项, 如 `tabstop`, 我们用 `&tabstop` 直接读到的是本地值了, 要访问全局值就必须使用 `&g:tabstop`。

`&l`: 表示访问一个选项的本地 (local) 值。对于有本地值的选项, 如 `tabstop`, 我们用 `&tabstop` 直接读到的已经是本地值了, 但修改则和 `set` 一样, 同时修改本地值和全局值。使用 `&l`: 前缀可以允许我们仅修改本地值, 像 `setlocal` 命令一样。

你可能要问, 什么时候我们会需要用变量形式来访问选项, 而不是使用 `set`、`setlocal` 这样的命令呢? 答案是, 当我们需要计算出选项值的时候。`set filetype=cpp` 基本上和 `let &filetype = 'cpp'` 等效, 我们需要注意到后者里面 `cpp` 是个字符串, 可以通过某种方式算出来的。光使用 `set`, 就不方便做到这样的灵活性了。

重要命令

Vim 里有很多命令, 很多我们已经介绍过, 或者直接在 `vimrc` 配置文件里使用了。这节里我们会介绍跟 Vim 脚本相关性比较大的一些命令。

首先是 `execute` (缩写 `exe`), 它能用来把后面跟的字符串当成命令来解释。跟上一节使用选项还是 `&` 变量一样, 这样做可以增加脚本的灵活性。除此之外, 它还有两种常见特殊用法:

在使用键盘映射等场合、需要在一行里放多个命令时, 一般可以使用 `|` 来分隔, 但某些命令会把 `|` 当成命令的一部分 (如 `!`、`command`、`nmap` 和用户自定义命令), 这种时候就可以使用 `execute` 把这样的命令包起来, 如: `exe '!ls' | echo 'See file list above'`。

`normal` 命令把后面跟的字符直接当成正常模式命令解释，但如果其中包含有特殊字符时就不方便了。这时可以用 `execute` 命令，然后在 " 里可以使用转义字符。我们上面讲字符串时没说的是，按键也可以这样转义，比如，"`\<C-W>`" 就代表 `Ctrl-W` 这个按键。所以，如果你想在脚本中控制切换到下一个窗口，可以写成：`exe "normal \<C-W>w"`。

然后，我要介绍一下 `source` (缩写 `so`) 命令。它用来载入一个 Vim 脚本文件，并执行其中的内容。我们已经多次在 `vimrc` 配置文件中用它来载入系统提供的 Vim 脚本了，如：

[复制代码](#)

```
1 source $VIMRUNTIME/vimrc_example.vim
2 ...
3 command! PackUpdate packadd minpac | source $MYVIMRC | call minpac#update('',
4 ...
```

这里要注意的地方是，要允许一个文件被 `source` 多次，是需要一些特殊处理的。我目前给出的 `vimrc` 配置文件由于需要被载入多次，进行了下面的特殊处理：

清除缺省自动命令组里当前的所有命令，以免定义的自动命令被执行超过一次

使用 `command!` 来定义命令，避免重复命令定义的错误

使用 `function!` 来定义函数，避免重复函数定义的错误

没有手工设置 `set nocompatible`，因为该设置可能会有较多的副作用（在 `defaults.vim` 里会确保只设置该选项一次）

上面我已经展示了一个 `command` 命令的例子。这个命令允许我们自定义 Vim 的命令，并允许用户来定制自动完成之类的效果（详见 [`:help user-commands`](#)）。注意这个命令的定义要写在一行里，所以如果命令很长，或者中间出现会吞掉 | 的命令的话，我们就会需要用上 `execute` 命令了。

最后，我再说明一下我们用过的 `map` 系列键映射命令（详见 [`:help key-mapping`](#)）。这些命令的主干是 `map`，然后前面可以插入 `nore` 表示键映射的结果不再重新映射，最前面用 `n`、`v`、`i` 等字母表示适用的 Vim 模式。在绝大部分情况下，我们都会

使用带 `nore` 这种方式，表示结果不再进行映射（排除偶尔偷懒的情况）。但是，如果我们的 `map` 命令的右侧用到了已有的（如某个插件带来的）键映射，我们就必须使用没有 `nore` 的版本了。

事件

和用户主动发起的命令相对应，Vim 里的自动处理依赖于 Vim 里的事件。迄今为止，我们已经遇到了下面这些事件：

`BufNewFile` 事件在创建一个新文件时触发

`BufRead`（跟 `BufReadPost` 相同）事件在读入一个文件后触发

`BufWritePost` 事件在把整个缓冲区写回到文件之后触发

`FileType` 事件在设置文件类型（`filetype` 选项）时被触发

Vim 里的事件还有很多（详见 [`:help autocmd-events-abc`](#)），我们就不一一介绍了。上面这些是我们最常用的，你应该了解它们的意义。

内置函数

Vim 里内置了很多函数（列表见 [`:help function-list`](#)），可以实现编程语言所需要的基本功能。我们目前用得比较多的是下面这两个：

`exists` 用来检测某一符号（变量、函数等）是否已经存在。在 Vim 脚本里最常见的用途是检测某一变量是否已经被定义。

`has` 用来检测某一 Vim 特性（列表见 [`:help feature-list`](#)）是否存在。帮助文档里已经描述得很清楚，我就不详细介绍了。你可以对照看一下我们的 `vimrc` 配置文件里的用法，应该就明白了。

Vim 的内置函数真的很多，我也没法一一介绍。你可以稍作浏览，了解其大概，然后在使用中根据需要查询。别忘了，在看 Vim 脚本时，在关键字上按下 `K` 就可以查看这个关键字的帮助，如下图所示：

```

" Enable file type detection.
" Use the default filetype settings, so that mail gets 'tw' set to 72,
" 'cindent' is on in C files, etc.
" Also load indent files, to automatically do language-dependent indenting.
" Revert with ":filetype off".
filetype plugin indent on

" Put these in an autocmd group, so that you can revert them with:
" ":augroup vimStartup | au! | augroup END"
augroup vimStartup
au!

" When editing a file, always jump to the last known cursor position.
" Don't do it when the position is invalid, when inside an event handler
" (happens when dropping a file on gvim) and for a commit message (it's
" likely a different one than last time).
autocmd BufReadPost *
  \ if line("'\"") >= 1 && line("'\"") <= line("$") && &ft !~# 'commit'
  \ |   exe "normal! g`\""
  \ | endif

augroup END

endif

" Convenient command to see the difference between the current buffer and the
" file it was loaded from, thus the changes you made.
" Only define it when not defined already.
" Revert with: ":delcommand DiffOrig".
if !exists("DiffOrig")
  command DiffOrig vert new | set bt=nofile | r ++edit # | 0d_ | diffthis
  \ | wincmd p | diffthis
endif

```

<MacVim.app/Contents/Resources/vim/runtime/defaults.vim [utf-8] 123,7 94%

在 Vim 脚本里使用 K 键查看帮助

风格指南

结束 Vim 脚本的介绍之前，我向你推荐一下 Google 出品的 Vim 脚本风格指南，
[Google Vimscript Style Guide](#)。写一种语言，有一个风格指南肯定是会有帮助的，尤其对于初学者而言。

Python 集成（选学）

Vim 脚本功能再强大，也还是一种小众的编程语言。所以，Vim 里内置了跟多种脚本语言的集成，包括：

Python

Perl

Tcl

Ruby

Lua

MzScheme


由于 Python 的高流行度，目前 Vim 插件里常常见到对 Python 的要求——至少我还没有用过哪个插件要求有其他语言的支持。所以，在这儿我就以 Python 为例，简单介绍一下 Vim 对其他脚本语言的支持。各个语言当然有不同的特性，但支持的方式非常相似，可以说是大同小异。

这部分作为选学提供，相当于本讲内部的一个小加餐。Python 程序员一定要把这部分读完，其他同学则可以选择跳到内容小结。

Vim 很早就支持了 Python 2，Vim 的命令 `python`（缩写 `py`）就是用来执行 Python 2 的代码的。后来，Vim 也支持了 Python 3，使用 `python3`（缩写 `py3`）来执行 Python 3 的代码。鉴于 Python 的代码还是有不少是 2、3 兼容的，Vim 还有命令 `pythonx`（缩写 `pyx`）可以自动选择一个可用的 Python 版本来执行。

我在 [🔗 拓展 3](#) 里给出了一段代码，用 Python 来检测当前目录是不是在一个 Git 库里。我们先用 `pythonx` 命令定义了一个 Python 函数，然后用 `pyxeval` 函数来调用该函数。这就是一种典型的使用方式：在 Python 里定义某个功能，然后在 Vim 脚本里调用该功能。这种情况下，Python 部分的代码一般不需要对 Vim 有任何特殊处理，只是简单实现某个特定功能。

下面是另一个小例子，通过 Python 来获得当前时区和协调世界时的时间差值（对于中国，应当返回 `_+0800`）：

 复制代码

```
1 function! Timezone()  
2     if has('pythonx')  
3         pythonx << EOF  
4         import time  
5  
6         def my_timezone():
```

```
7     is_dst = time.daylight and time.localtime().tm_isdst
8     offset = time.altzone if is_dst else time.timezone
9     (hours, seconds) = divmod(abs(offset), 3600)
10    if offset > 0: hours = -hours
11    minutes = seconds // 60
12    return '{:+03d} {:02d}'.format(hours, minutes)
13 EOF
14     return ' ' . pyxeval('my_timezone()')
15 else
16     return ''
17 endif
18 endfunction
```

从 `pythonx << EOF` 到 `EOF`，中间是 Python 代码，定义了一个叫 `my_timezone` 的函数，我们然后调用该函数来获得结果。对于不支持 Python 的情况，我们就直接返回一个空字符串了。

另一种更复杂的情况是，我们的主干处理逻辑就放在 Python 里。这种情况下，我们就需要在 Python 里调用 Vim 的功能了。在 Vim 调用 Python 代码时，Python 可以访问 `vim` 模块，其中提供多个 Vim 的专门方法和对象，如：

`vim.command` 可以执行 Vim 的命令

`vim.eval` 可以对表达式进行估值

`vim.buffers` 代表 Vim 里的缓冲区

`vim.windows` 代表当前标签页里的 Vim 窗口

`vim.tabpages` 代表 Vim 里的标签页

`vim.current` 代表各种 Vim 的“当前”对象（详见 [`:help python-current`](#)），包括行、缓冲区、窗口等

此外，在拓展 2 里我们给出的使用 `pyxf` 来执行一个 Python 脚本文件，也是一种在 Vim 里调用 Python 的方式（详见 [`:help pyxfile`](#)）。那段 `clang-format` 的代码，总体上也就是访问 `vim.current.buffer` 对象，调用外部命令格式化指定行，然后把修改的内容写回到 Vim 缓冲区里。

内容小结

好了，我们的 Vim 脚本介绍就到这里了。这一讲和大部分其他讲不同，只是给了你一个 Vim 脚本的概览，目的是让你全面了解一下 Vim 脚本，能够读懂一般的 Vim 脚本，而不是真正教会你如何去写脚本。这讲的主要知识点是：

Vim 脚本的基本语法，包括变量、数字、字符串、复杂数据结构、表达式、控制结构和函数

Vim 的专门特性，包括变量的前缀、脚本相关命令、Vim 里的事件和内置函数

Vim 脚本风格指南

Vim 对 Python 等其他脚本语言的支持

作为一门编程语言，只有在实践中不断操练，才能真正学会它的使用。如果你对 Vim 脚本有兴趣的话，我们下一讲会剖析几个 Vim 脚本来分析一下，让你有更深入的体会。

课后练习

请查看几个现有的 Vim 脚本来仔细分析一下，理解各行的意义。建议可以从我们在 vimrc 配置文件中包含的 vimrc_example.vim 开始，然后查看其中使用的 defaults.vim。别忘了，我们可以使用普通模式快捷键 gf 或 <C-W>f 直接跳转到光标下的文件里。

如果遇到什么问题，欢迎留言和我讨论。我们下一讲再见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 13 | YouCompleteMe: Vim 里的自动完成

下一篇 15 | 插件荟萃：不可或缺的插件

精选留言 (2)

写留言



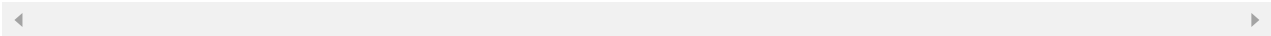
我来也
2020-08-31

干货满满!
照着文章学,看懂基本的vim脚本是没问题了.

看到vim的匿名函数,想不到vim脚本居然还支持函数式编程.

...
展开

作者回复: 这一讲又继续赶上来坐沙发啦。💎💎💎💎💎



2



瀚海星尘
2020-10-23

和老师一起写了这么久的配置，可算是能看懂了，哈哈！



1