Vim 編程指北

著
水龍夢寰

指南是細則　指北是原則

Vim大師煉成記

VimL 語言編程指北

水龍、夢寰

2023.02.04

前言

這是一篇有關 Vim 腳本語言的入門與進階教程。是 “指北”，不是 “指南”，所以如果不慎指錯了路，切勿見怪。不過要相信地球是圓的，繞了一圈之後，希望還是能找對目標的。

初學者如果第一章看不懂，建議直接看第二章；如果第二章看不懂，建議直接看第三章；如果第三章也看不懂，建議直接放棄治療，汝須先培養對 vim 的信仰習慣。

以下 …… 開始嚴肅話題。

正名約定

Vim 是上古神器之一，且能歷久彌新，與時俱進。隨着 Vim 的發展，Vim 的腳本也逐漸發展壯大，支持的功能與特性越來越豐富，儼然成爲一種新的或老的實用腳本語言。然而這語言的名字，網上的稱謂似乎還有點五花八門。

爲了行文統一與方便，在這裏我採用 “VimL” (Vim Language 縮寫) 來表示該腳本語言，用 “Vim” 表示編輯器；而小寫的 “vim” 則是指系統中可執行的編輯器程序，若從 VimL 角度看，也算是它的解釋器程序；然後 vim script 就是存放 VimL 代碼且可用 vim 解釋運行它的文本文件。

目標假設

本教程針對的目標羣體，假定是有使用 Vim 的基礎及一定的編程基礎。儘管我儘量從基本概念講起，但一些最基礎的東西怕無從再細緻了。然後最重要的是要熱愛 Vim ，並且有折騰的精神來打造或調教自己的 Vim。

其實，不管是使用 Vim 還是 VimL，最好的資源都是 Vim 的內置幫助文檔（:help）。外部教程都不免有所側重，較適於學習階段的引領者。

本教程依據的 Vim 版本是 8.1，系統環境 Linux。但除了一些新特性，應該也適用 Vim7 以下版本。同時由於 Vim 本身是跨平臺的，VimL 自然也與操作系統無關。雖然無法一一驗證，但在一些重要的差異處，儘量在文中指出。

最新的 Vim 版本是 9，主要是大幅提升性能，同時也改變了之前版本的部分用法，整體向現代的通用腳本語言靠近。本書後續計劃加入 Vim 9 的新語法，但因本書講的更多是通用的基礎且 Vim 9 還在不斷發展，待時機成熟後再更新也不遲。

VimL 的優缺點

作爲一種語言，首先指出 VimL 的缺點。一是隻能在 Vim 環境下運行，二是運行速度有點慢。但是，對於熱衷 Vim 的程序猿，每天的編碼工作都在 Vim 環境下，VimL 的編程方式與 Vim 的操作方式無間密合，應該算是個優勢。

另外，程序的運行速度都是相對的。所有的動態腳本語言，相對靜態的編譯語言，都很慢。但這不要緊，只要完成大部分工作能足夠快，腳本的簡單便捷性就能體現出來了。VimL 同樣具有腳本語言的這個共性。

用 Vim 編寫 VimL 代碼，另有個天然的優勢，就是編輯器，解釋器，與文檔手冊一體化，同時仍然保持了 Vim 的小巧，不像靜態語言的 IDE 那麼笨重。

編程思想基本是獨立於語言的，大多數語言都是相通的。現代的高級腳本語言更是幾乎都能提供差不多的功能。(而且，據說只要是“圖靈完備”的語言，理論上都能做任何事)。所以，經常使用 Vim 的程序猿，如果想多學一門腳本語言，那 VimL 是個不壞的選擇。

文本約定

帶星號‘*’的章節，表示略有艱深晦澀的內容，可以選擇性略過。

關於示例代碼塊，‘:’開始的行表示 Vim 的命令行（也叫 ex 命令），‘$’開始的行表示從 shell 運行的命令行。較短的示例代碼，可以直接輸入或粘貼入 vim 的命令行，較長的示例代碼，建議保存‘.vim’文件，然後‘:source’。所有正文中出現的 ex 命令都以單引號 ‘ 括起來，部分內容若需要，也可能用單引號 ‘ 括起來，除此之外還需要括號括起來的內容都用雙引號“”。

Vim 代碼和 Shell 命令行命令均用方框括起來了，並輔以了淺綠色背景。使用綠色是因爲 Vim 的 Logo 就是綠色的，此外綠色也是環保色。此綠色具體的 RGB 值爲 {1,152,0}，灰度爲 35%。

本書正文共十章，在行文組織上儘量循序漸進，可粗略分爲三部分，建議按順序閱讀。第 1-3 章爲基礎篇，第 4-7 章爲中級篇，第 8-10 爲高級篇。最後的附錄是 Vim 常用命令及鍵位總結。文中經常提示用‘:help’命令查閱相關幫助主題，此後忘記細節時可隨時查詢。

水龍、夢寰

目錄

第一章　VimL 語言主要特點

1.1　Hello World 的四種寫法

按慣例，我們討論一門語言，首先看下如何寫最簡單的“Hello World”（程序）。由於 Vim 是高度自由的，VimL 也有多種不同的方式玩轉“Hello World”。

速觀派：直接操起命令行

最快速的辦法是在 Vim 命令行下用‘:echo’命令輸出“Hello World”：

```
: echo 'Hello World!'
```

唯一需要注意的是，得把“Hello World”用引號括起來，單引號或雙引號都可以。這樣再按下回車就能在 Vim 的消息區顯示出“Hello World”這行字符串了。

由於這條消息字符串很簡短，一行就能顯示完整，Vim 將其直接顯示在命令行的位置，並且運行完直接返回 Vim 普通模式。如果字符串很長或多行字符串，則消息區將向上滾動，以顯示完整的消息，用戶需要額外按個回車纔回普通模式。

試試在命令行輸入這條命令，看看有啥不同反應：

```
: echo "Hello World! \n Hello World! \n Hello World!"
```

好了，你已經學會了如何用 VimL 輸出“Hello World”了。這也算編程嗎？別逗了！其實，別把編程想得那麼嚴肅，那這就算編程！

正規派：建立腳本文件

把剛纔在命令行輸入的那條命令保存在一個“.vim”後綴的文本文件中，那就是一個 vim script 了，這是用 VimL 編程比較正常的用法。

爲了方便，建議在本地建個目錄，用於保存本教程的示例代碼。比如：

```
$ cd ~/.vim
$ mkdir vimllearn
$ vim vimllearn/hello1.vim
```

這將在 ~/.vim 目錄下新建一個 vimllearn 目錄，並用 vim 開始編輯一個名爲 hello1.vim 的文件。vim 會爲該文件新建一個緩衝區 buffer，在該 buffer 中輸入以下文本，然後輸入命令‘:w’保存：

```
" 文件：hello1.vim
" 用途：VimL hello world 示例
" 作者：lymslive、shieber
" 時間：2017-08、2022-05

echo 'Hello World!'

finish

脚本結束了，可以隨便浪~~
不管寫些什么亂七八糟的都木有問題。
```

你其實可以只在該文件中寫入 echo ‘Hello World!’ 這一行就夠了。幾點說明：

1. 前面以一個雙引號” 開始的行是註釋，註釋也可以寫在行尾。

2. 在脚本文件中，echo 命令前不必加冒號‘:’，但是加上冒號也是允許的。

3. finish 表示直接結束脚本，在之後的語句都不再被 vim 解析；這是可選的，沒有遇到 finish 就會執行到文件最後一行。

當有了‘*.vim’脚本文件，就可以在 vim 環境中用‘:source’命令加載運行了：

```
: source ~/.vim/vimllearn/hello1.vim
```

需要將脚本文件的路徑寫在 source 命令之後作爲參數。如果當前 vim 正常編輯 hello1.vim 這個文件，則可用% 表示當前文件的路徑：

```
: source %
```

折騰並解釋了這許久，終於可以通過 source 一個 vim 脚本打印輸出 “Hello World” 了，與此前的效果是一樣一樣的。當然了，用 VimL 寫脚本肯定不能只滿足於寫 “Hello World” 吧，但是這纔是標准用法。

此外 Vim 的命令是可以簡寫的，source 可簡寫爲 so。當你在寫一個 vim 脚本時想快速驗證執行該脚本時，可以只輸入：

```
: so %
```

如果還想更省鍵，就定義一個快捷鍵映射吧，比如：

```
: nnoremap <F5>:update<CR>:source %<CR>
```

可以將這行定義加入你的 vimrc 中，不過最好是放在 ~/.vim/ftplugin/vim.vim 中，並加上局部參數，讓它隻影響‘*.vim’文件：

```
: nnoremap <buffer> <F5>:update<CR>:source %<CR>
```

測試派：進入 Ex 模式

直接在命令行用‘:echo’查看一些東西其實很有用的，可以快速驗證一些記不清楚的細節。比如你想確認下在 VimL 中字符‘0’是不是與數字 0 相等，可以這樣：

```
: echo '0' == 0
```

但如果要連續輸入多條命令並查看結果，每次都要（從普通模式）先輸入個冒號，不免有些麻煩。這時，Ex 模式就有用了。默認情況下（若未在 vimrc 被改鍵映射），在普通模式下用 Q 鍵進入 Ex 模式。例如，在 Ex 模式下嘗試各種輸出“Hello World”的寫法，看看不同引號對結果的影響：

```
Entering Ex mode.  Type "visual" to go to Normal mode.
: echo 'Hello World!'
: echo "Hello World!"
: echo 'Hello \t World! \n Hello \t World!'
: echo "Hello \t World! \n Hello \t World!"
: vi
```

最後，按提示用 visual 或簡寫 vi 命令回到普通模式。

Vim 的 Ex 模式有點像 VimL 的交互式的解釋器，不過語法完全一樣（有些腳本語言的交互式解釋器與執行腳本有些不同的優化），仍然要用 echo 顯示變量的值。

* 索隱派：從 shell 直接運行

如果只爲了運行一個 vim script 腳本，也不一定要先啓動 vim 再 source，直接在啓動 vim 時指定特定參數也能辦到。‘-e’參數表示以 Ex 模式啓動 vim，‘-S’參數啓動後立即 source 一個腳本。因此，也可以用如下的命令來輸出“Hello World”：

```
$ cd ~/.vim/vimllearn/
$ vim -eS hello1.vim
```

這就相當於使用 vim 解釋器來運行 hello.vim 這個腳本，並且停留在交互式界面上。此時可以用‘:q’命令退出，或‘:vi’以正常方式繼續使用 Vim。

vim 本身的命令行啓動參數其實還支持很多功能，請查閱‘:help starting’。其中還有個特殊的參數是‘-s’，如果與‘-e’聯用，就啓動靜默的批處理模式，試試這個：

```
$ vim -eS hello1.vim -s
```

沒有任何反應輸出？因爲‘-s’使普通的 echo 提示無效，看不到任何提示！趕緊輸入 q 回車退出 vim 回到 shell。因爲如果不小心按了其他鍵，vim 可能就將其當作命令來處理了，而且不會有任何錯誤提示，這就會讓大部分人陷入不知如何退出 vim 的恐慌。

雖然‘vim -e -s’不適合來輸出 “Hello World”，但如果你的腳本不是用來做這種無聊的任務，這種模式還是有用的。比如批處理，在完全不必啓動 vim 可視編輯的情況下，批量地對一個文件或多個文件執行編輯任務。可達到類似 sed 的效果。而且，在 vim 腳本寫好的情況下，不僅可以按批處理模式執行，也可以在正常 Vim 可視編輯某個文件時，遇到類似需求時，也可以再手動‘:source’腳本處理。

小結

運行 vim 腳本的常規方法用‘:source’命令，而且有很多情況下並不需要手動輸入‘:source’命令，在滿足一定條件下，vim 會自動幫你 source 一些腳本。vim 的啓動參數‘-S’其實也是執行‘:source’。

Vim 的命令行可以隨時手動輸入一些簡短命令以驗證某些語法功能，進入 Ex 模式則可以連續手動輸入命令並執行。Ex 模式雖然比較少用，但不該反感排斥，這對學用 VimL 還是大有裨益的，以後會講到，VimL 的 debug 功能也是在 Ex 模式中的。

靜默批處理‘vim -e -s’本質上也是 Ex 模式，不過禁用或避免了交互的中斷。屬於黑科技，一般的 vim 用戶可先不必深究。

* 拓展閱讀：Vim 與可視化

“可視化”是個相對的概念。現在說到可視化，似乎是指功能豐富的 IDE 那種，有很多輔助窗口展示各方面的信息，甚至有圖形化來表示類層次關係與函數調用關係。還有傳說中的唯一的中文編程語言“易語言”還支持圖文框拖拖拽拽就能編寫代碼的東東……而 vim 這種古董，只有編輯純文本，似乎就該歸屬於“不可視”。

然而，讓我們回顧洪荒時代，體驗一下什麼叫真正的“不可視”編輯。

在 Vi 都還沒誕生的時代，有一個叫 ed 的行編輯器，它只能通過命令以行爲單位去操作或編輯文本文件。它完全沒有界面，無從知道當前編輯的是哪個文件，在哪行，當前行是什麼內容，用戶只能“記住”，或用命令查詢。比如用 p 命令打印顯示當前行（不過也可以在前面帶上行地址打印多行，至今 vim 的大部分命令都可以帶地址參數）。要編輯當前行，請用 a、i 或 c 命令（vimer 有點眼熟吧），不過編輯完後也無從知曉結果是否正確，可能還需要再用 p 命令打印查看確證。

之後，有個 ex 編輯器，不過是對 ed 的命令進行了擴展，本質上仍是行編輯器。直到 vi 橫空出世，那才叫“屏幕編輯器”。意思是可以全屏顯示文件的許多行，移動光標實時修改某一可見行，修改結果即時顯示……這纔像我們現在可認知的編輯器了。

然後是 vim 對 vi 的擴展增強。事實上，vim 還不如 vi 的劃時代意義，它的增強與此前 ex 對 ed 的增強是差不多的程度，基本上是平行擴展。

可視化程度不是越高越好。vim 與 vi 都保留與繼承了 ex 的命令，因爲 ex 命令確實

高效，當你能確知一個命令的運行結果，就沒必要關注中間過程了。比如最平凡無奇但常用的 ex 命令就是 “:s” 全局替換命令。

VimL 語言就是基於 ex 命令的，再加上一些流程控制，就成了一種完整的腳本語言。如果說 vim 對 vi 有什麼壓倒性的里程碑意義，我覺得應是豐富完善了 VimL 語言，使得 vim 有了無窮的擴展與定製。利用 VimL 寫的插件，既可以走增加可視化的方向，也可以不增加可視化而偏向自動化。依每人的性格習慣不同，可能會在 Vim 的可視化與自動化之間找到適合自己的不同的平衡點。

1.2 同源 ex 命令行

那麼，VimL 到底是種什麼樣的語言。這裏先說結論吧，VimL 就是富有程序流程控制的 ex 命令。用個式子來表示就是：

```
VimL = ex 命令 + 流程控制
```

VimL 源於 ex，基於 ex，即使它後來加了很多功能，也始終兼容 ex 命令。

然則什麼是 ex 命令，這不好准確定義，形象地說，就是可以在 Vim 底部命令行輸入並執行的語句。什麼是流程控制，這也不好定義呢，類比地說，就是像其他大多語言支持的選擇、循環分支，還有函數，因爲函數調用也是種流程跳轉。

下面，還是用些例子來闡述。

第一個腳本：vimrc

爲了說明 vimrc 先假設你沒有 vimrc。這可以通過以下參數啓動 vim：

```
$ cd ~/.vim/vimllearn/
$ vim -u NONE
```

這樣啓動的 vim 不會加載任何配置文件，可以假裝自己是個只會用裸裝 vim 的萌新。同時也保證以下示例中所遇命令沒有被重映射，始終能產生相同的結果。

Vim 主要功能是要用來編輯一些東西的，所以我們需要一些語料文本。這也可以用 vim 的普通命令生成，請在普通模式下依次輸入以下按鍵（命令）：

```
20aHello World!<ESC>
yy
99p
: w helloworld.txt<CR>
```

其中輸入的按鍵不包括換行符，上面分幾行顯示，只爲方便分清楚幾個步驟的命令。

首先是個 a 命令，進入插入模式，輸入字符串 “Hello World!”，然後按 <ESC> 鍵返回普通模式（這裏 <ESC> 表示那個衆所周知的特殊鍵，不是五個字符啦）。a 之前的 20 是還在普通模式下輸入的數字參數，（它不會顯示在光標所在的當前行，而是臨時顯示在右

下角，然而一般不必關注）這表示後來的命令重複執行多少次。所以結果是在當前行插入了 20 個 “Hello World!”，也就是新文件的第一行。

接着命令 yy 是複製當前行，99p 是再粘貼 99 行。於是總共得到 100 行 “Hello World!” ——滿屏盡是 Hello World!，應該相當壯觀。

最後的命令是用冒號 ‘:’ 進入 ex 命令行，保存文件，<CR> 表示回車，ex 命令需要回車確認執行。

現在，我們已經在用 vim 編輯一個名爲 helloworld.txt 的文件了。看着有點素是不是? 可以用下面的 ex 命令設置行號選項：

```
: set number
```

如此就會在文本窗口左則增加幾列特殊列，爲文件中的每行編號，確認一下是不是恰好 100 行，用 G 普通命令翻到最後一行。

還有，是不是覺得每一行太長了，超過了窗口右邊界。(如果你用的是超大顯示屏，Vim 的窗口足夠大還沒超過，那麼在一開始的示例中，把數字參數 20 調大吧）如果想讓 vim 折行顯示，則用如下命令設置選項：

```
: set wrap
```

可以看到，長行都折行顯示了，但是行編號並沒有改變。也就是說文件中仍是隻有 100 行，只有太長的行，vim 自動分幾行顯示在屏幕窗口上了。

你可以繼續輸入些設置命令讓 vim 的外觀更好看些，或讓其操作方式更貼心些。但是等等，這樣通過冒號一行行輸入實在是太低效，應該把它保存到一個 vim 腳本文件中。

按冒號進入命令行後，再按 <Ctrl-F> 將打開一個命令行窗口，裏面記錄着剛纔輸入的 ex 歷史命令。這個命令窗口的設計用意是爲了便於重複執行一條歷史命令，或在某條歷史命令的基礎上小修後再執行。不過現在我們要做的是將剛纔輸入的兩條命令保存到一個文件中，比如就叫 vimrc.vim，整個按鍵序列是：

```
: <Ctrl-F>
Vk
: '<, '> w vimrc.vim<CR>
: q<CR>
```

解釋一下：進入命令窗口後光標自動在最後一行，V 表示進入行選擇模式，k 上移一行，即選擇了最後兩行。在選擇模式下按 ‘:’ 進入命令行，會自動添加 ‘<, ’ >，這是特殊的行地址標記法，表示選區，然後用 ‘:w’ 命令將這兩行寫入 vimrc.vim 文件（注意當前目錄應在 ~/.vim/vimllearn 中）。最後的 ‘:q’ 命令只是退出命令窗口，但 vim 仍處於編輯 helloworld.txt 狀態中。

你需要再輸入一個 ‘:q’ 退出 vim，然後用剛纔保存的腳本當作啓動配置文件重新打開文件，看看效果：

```
:q<CR>
```

```
$ vim -u vimrc.vim helloworld.txt
```

可見，重新打開 helloworld.txt 文件後也自動設置了行號與折行。你可以換這個參數啓動 vim 對比下效果，確認是 vimrc.vim 的功效：

```
$ vim -u NONE helloworld.txt
```

可以手動編輯 vimrc.vim 增加更多配置命令：

```
: e vimrc.vim
```

這樣就切換到編輯‘:qvimrc.vim’狀態了，裏面已經有了兩行，用普通命令 Go 在末尾打開新行進入插入模式，加入如下兩行（還可自行添加一些註釋）：

```
: nnoremap j gj
: nnoremap k gk
```

按 <ECS> 回普通模式再用‘:w’保存文件。可以退出 vim 後重新用 $ vim -u vimrc.vim helloworld.txt 參數啓動 vim 打開文件觀察效果。也可以在當前的 vim 環境中重新加載 vimrc.vim 令其生效：

```
: source %
: e #
```

其中，‘:e #’或 <Ctrl-^> 表示切換到最近編輯的另一個文件，這裏就是 helloworld.txt 文件啦，在這個文件上移動 j k 鍵，看看是否有什麼不同體驗了。

不過，表演到此爲止吧。這段演示示例主要想說明幾點：

1. VimL 語言沒什麼神祕，把一些 ex 命令保存到文件中就是 vim 腳本了。

2. vimrc 配置文件是 vim 啓動時執行的第一個腳本，也應是大多數 Vim 初學者編寫的第一個實用腳本。

關於 vim “命令”這個名詞，還有一點要區分。普通模式下的按鍵也叫 “命令”，可稱之爲 “普通命令”，但由於普通模式是 Vim 的主模式，所以 “普通命令” 也往往簡稱爲 “命令” 了。通過冒號開始輸入而用回車結束輸入的，叫 “ex 命令”，vim 腳本文件不外是記錄的 “ex 命令” 集。（注：宏大多是記錄普通命令）

默認的 vimrc 位置

正常使用 vim 時不會帶 -u 啓動參數，它會從默認的位置去找配置文件。這可以在 shell 中執行這個命令來查看：

```
$ vim --version
```

或者在任一已啓動的 vim 中用這個 ex 命令‘:version’也是一樣的輸出。在中間一段應該有類似這樣幾行：

```
系統 vimrc 文件: "$VIM/vimrc"
用户 vimrc 文件: "$HOME/.vimrc"
第二用户 vimrc 文件: "~/.vim/vimrc"
用户 exrc 文件: "$HOME/.exrc"
defaults file: "$VIMRUNTIME/defaults.vim"
```

它告訴了我們 vim 搜索 vimrc 的位置與順序。主要是這兩個地方,~/.vimrc,~/.vim/vimrc。用戶可用其中一個做爲自定義配置，強烈建議用第二個 ~/.vim/vimrc。因爲配置可能漸漸變得很複雜，將所有配置放在一個目錄下管理會更方便。不過有些低版本的 vim 可能不支持 ~/.vim/vimrc 配置文件，在 unix/linux 系統下可將其軟鏈接爲 ~/.vimrc 即可。

需要注意的是，vimrc 是個特殊的腳本，習慣上沒有‘.vim’後綴。

如何配置 vimrc 屬於使用 Vim 的知識（或經驗）範疇，不是本 VimL 教程的重點。不過爲了說明 VimL 的特點，也給出一個簡單的示例框架如下：

```
" File: ~/.vim/vimrc

let $VIMHOME = $HOME . '/.vim'
if has('win32') || has ('win64')
    let $VIMHOME = $VIM . '/vimfiles'
endif

source $VIMHOME/setting.vim
source $VIMHOME/remap.vim
source $VIMHOME/plug.vim

if has('gui')
    " source ...
endif

finish
let $USER = 'vimer'
echo 'Hello '.$USER '! Working on: '.strftime("%Y-%m-%d %T")
```

一般地，一份 vimrc 配置包括選項設置，快捷鍵映射，插件加載等幾部分，每部分都可能變得複雜起來，爲方便管理，可以分別寫在不同的 vim 腳本中，然後在主 vimrc 腳本中用‘:source’命令調用。這就涉及腳本路徑全名問題了，若期望能跨平臺，就可創建一個變量，根據運行平臺設置不同路徑，這就用到了‘:if’分支命令了。

最後兩行打印個歡迎詞。你可以將自己的大名賦給變量 $USER。如果，你覺得這很傻很天真，可以移到 finish 之後就不生效了。

在 vimrc 中，選擇分支可能很常見，根據不同環境加載合適的配置嘛。但循環就很少見了。因爲 vim 向來還有個追求是小巧，啓動快，那麼你在啓動腳本中寫個循環是幾個意思啊，萬一寫個死循環 BUG 還啓不起來了。

流程控制語句也是 ex 命令

在 VimL 中，每一行都是 ex 命令。作爲一門腳本語言，最常見的，創建變量要用‘:let’命令，調用函數要用‘:call’命令。初學者最易犯與迷惑的錯誤，就是忘了 let 或 call，裸用變量，裸調函數，比如：

```
1 i = -1
2 abs(-1)
```

用過其他語言的可能會覺得這很自然，但在 VimL 中是個錯誤，因爲它要求第一個詞是欽定的 ex 命令！正確的寫法是：

```
1 let i = -1
2 call abs(-1)
```

從這個意義上講，VimL 與 shell 腳本很類似的，把命令行語句保存到文件中就成了腳本。每一行都以可執行命令開始，後面的都算作該命令的參數。

在 VimL 中，‘:if’、‘:for’、‘:while’也當作擴展的 ex 命令處理，在 vim 腳本中，這些“關鍵詞”前面，可以像‘:set’一樣加個可選的冒號。同時，也可以像其他 ex 命令一樣在無歧義時任意簡寫。比如：

- :endif 可簡寫爲 en 或 end 或 endi 或 endif
- :endfor 可簡寫爲 endfo
- :endfunction 可簡寫爲 endf，後面補上 unction 任意前幾個字符也可以。

這套縮寫規則，就與替換命令‘:substitute’簡寫爲‘:s’，設置命令‘:set’簡寫爲‘:se’一樣一樣的。但是，在寫腳本時，強烈建議都寫命令全稱。命令簡寫只爲在命令行中快速輸入，而在腳本中只要輸入一次，一勞永逸，就應以可讀性爲重了。

當有了這個意識，VimL 的一些奇怪語法約定，也就顯得容易理解多了。比如：

- ex 命令以回車結束，所以 VimL 語句也按行結束，不要在末尾加分號，加了反而是語法錯誤，在 Vim 中每個符號都往往有奇葩意義。
- VimL 的續行符 \寫在下一行的開始，其他一些語言是把 \寫在上一行結束，表示轉義掉換行符，合爲一行。但在 VimL 中，每一行都需要一個命令，你可以把 \想象爲一個特殊命令，意思是“合併到上一行”。
- 在 VimL 中，不推薦在一行寫多個語句，要寫也可以，把反斜槓 \扶正爲豎線 | 表示語句分隔吧。這在 vim 下臨時手輸單行命令時可能較爲常見，減少額外按回車與冒號。在很多鍵盤佈局中，| 與 \恰好在回車鍵上面。

關鍵命令列表

Vim 的 ex 命令集是個很大的集合，比絕大多數的語言的關鍵字都多一個數量級。幸運的是，我們寫 VimL 語言的腳本，並不需要掌握或記住這所有的命令，只要記住一些主要的關鍵命令就可以完成大部分需求了。

我從 VimL 語言的角度，按常用度與重要度並結合功能性將那些主要的命令分類如下，僅供參考：

```
1. let call (unlet)
2. if for while function try (endif, endfor, end...)
3. break continue return finish
4. echo echomsg echoerr
5. execute normal source
6. set map command
7. augroup autocmd
8. wincmd tabnext
9. 其他着重於編輯功能的命令
```

其中，第 5 類恰好是個分界線，之上的是形成 VimL 語言的關鍵命令，之下是作爲 Vim 編輯器的重要命令。沒有後面的編輯器命令，純 VimL 語言也可以寫腳本，作爲一種（沒有什麼優勢的）通用腳本而已；只有利用後面的編輯器命令，纔可以調控 Vim。

後面的編輯器命令也可以單獨使用，所以 Vim 高手也未必一定需要會 VimL。不過有了 VimL 語言命令的助力，那些編輯器命令可變得更高效與靈活。不過最後一大類純編輯命令，可能較少出現在 VimL 語言腳本中。因爲按 Vim 可視化編輯的理念，是需要使用者對這些編輯結果作出及時反饋的。同時，很多編輯命令也有相應的函數，在 VimL 中調用函數，可能更顯得像腳本語言。所以，VimL 中不僅有“庫函數”的概念，還有“庫命令”呢。

總之，語句與命令，是聯結 VimL 與 Vim 的重要紐帶。這是 VimL 語言的重要特點，也是初學者的一大疑難點。尤其是對有其他語言編程經驗的，可能還需要一定的思維轉換過程吧。

1.3 弱類型强作用域

“弱類型”不是 VimL 的特點，是幾乎所有腳本語言的特點。准確地說是變量無類型，但值有類型。創建變量時不必定義類型，直接賦值就行，也可以認爲是變量臨時獲得了值的類型。關於 VimL 的變量與類型，將在下一章的基礎語法中詳解。

變量作用域是編程的另一個重要概念，也幾乎每個語言都要管理的任務。這裏說 VimL 具有“强作用域”的特點，是指它提供了一種簡明的語法，讓用戶强調變量的作用域範圍。

VimL 語言級的作用域 g: l: s: a:

變量作用域的意義是指該變量在什麽範圍內可見，可被操作（讀值或賦值）。在 VimL 中，每個變量都可以加上一個冒號前綴，表示該變量的作用域。不過另有兩條規則：

1. 在一些上下文環境中，可以省略作用域前綴，等效於加上了默認的作用域前綴。

2. 有一些作用域前綴只在特定的上下文環境中使用。

從 VimL 語言角色看，主要有以下幾種作用域：

1. g: 全局作用域。全局變量就是在當前 vim 會話環境中，在任何腳本，任何 ex 命令行中都可以引用的變量。所有在函數之外的命令語句，都默認是全局變量。

2. l: 局部作用域。只可在當前執行的函數體內使用的變量，在函數體內的變量默認爲局部變量，l: 局部變量也只能在函數體內使用。

3. s: 腳本作用域。只有當前腳本內可引用的變量，包括該腳本的函數體內。

4. a: 參數作用域。特指函數的參數，在函數體內，要引用傳入的實參，就得加上 a: 前綴，但定義函數時的形參，不能加 a: 前綴。a: 還隱含一個限定是隻讀性，即不能在函數體內修改參數。

這幾種作用域前綴所對應的英文單詞，可認爲是 global、local、script 與 argument。不過 s: 也可認爲是 static，因爲在 C 語言中，static 也表示只在當前文件中有效的意思。

變量作用域的應用原則：

1. 儘量少用全局變量，因爲容易混亂使用，難於管理。不過在 ex 命令行或 Ex 模式下只爲臨時測試的語句，爲了方便省略前綴，是全局變量，當然在此命令中也只能是全局變量。在寫 vim 腳本文件時，若要使用全局變量，不要省略 g: 前綴。同時全局變量名儘量取得特殊點，比如全是大寫，或帶個插件名的長變量名，以減少被衝突的概率。

2. 局部變量的前綴 l: 一般可省略。但我建議也始終加上，雖然多敲了兩個字符，但編程的效率來源於思路清晰，不在於少那幾個字符。同時在 VimL 編程時，堅持習慣了作用域前綴，就能在頭腦中無形地加強這種意識，然後對作用域的把握也更加精准。另外，顯然地，在函數體內要引用全局就是必須加上 g: 前綴。

3. 在寫 vim 腳本時，函數外的代碼，能用 s: 變量就儘量用 s: 變量。對於比較大的腳本變量（如字典），想對外分享，也寧可先定義爲 s: 變量，再定義一個全局可訪問的函數來返回這個腳本變量。

4. 參數變量，a: 是語法強制要求，漏寫了 a: 往往是個錯誤，（如果它沒報錯，恰好與同名局部變量衝突了，那是更糟糕與難以覺察的錯誤）也是初寫 VimL 函數最容易犯的語法錯誤。

Vim 實體作用域 b: w: t:

Vim 作爲一個可視化的編輯器，給用戶呈現的，能讓用戶交互地操作的實體對象主要有 buffer（緩衝文件），window（窗口），tabpage（標籤頁）。可以把它們想象爲互有關係的容器：

- 緩衝對應着一個正在編輯中的文件，在不細究的情況下可認爲與文件等同。（不過不

一定對應着硬盤上的一個文件，比如新建的尚未保存的文件，以及一些特殊緩衝文件）緩衝也可認爲是容納着文件中所有文本行的容器，就像是簡單的字符串列表了。

• 窗口是用於展示緩衝文件的一部分在屏幕上的容器。Vim 可編輯的文件很大，極有可能在一個屏幕窗口中無法顯示文件的所有內容，所以窗口對應於緩衝文件還有個可視範圍。一個窗口在一個時刻只能容納一個緩衝文件，但在不同時刻可以對應不同的緩衝文件。

• 標籤頁是可以同時容納不同的窗口的另一層更大的容器。原始的 Vi 沒有標籤頁，標籤頁是 Vim 的擴展功能。標籤頁極大增強 Vim 的可視範圍，可認爲窗口是平面的，再疊上標籤頁就是（僞）立體的了。

• 一個緩衝文件可以展示在不同的窗口或（與）標籤頁中。所有已展示在某個窗口（包括在其他標籤頁的窗口）的緩衝文件都是“已加載”狀態，其他曾經被編輯過但當前不可見的緩衝文件則是“未加載”狀態，不過 Vim 仍然記錄着所有這些緩衝文件的列表。

然後，Vim 還有個“當前位置”的概念。也就是光標所在的位置，決定了哪個是“當前緩衝文件”，“當前窗口”與“當前標籤頁”。

有了這些概念，對 VimL 中的另外三個作用域前綴 b: w: t: 就容易理解了。其意即指一個變量與特定的緩衝文件、窗口或標籤頁相關聯的，以 b: 舉例說明。

• b:varname 表示在當前緩衝文件（實體對象）中存在一個名爲 varname 的變量。

• VimL 語句在執行過程中，只能直接引用當前緩衝文件的 b: 變量，如果要引用其他緩衝文件的變量，要麼先用其他命令將目標緩衝文件切換爲當前編輯的緩衝文件，或者調用其他的內置函數來訪問。

• 如果一個緩衝文件“消失”了，那麼與之關聯的所有 b: 變量也消失了。

• 窗口與標籤頁的“消失”能比較形象與容易地理解，關閉了就算消失了。但 Vim 內部對緩衝的管理比較複雜，未必是從窗口上不見了就代表“消失”了。

• 不過在一般 VimL 編程中，可暫不必深究緩衝文件什麼時候“消失”。只要記着一個 b: 變量必定與一個緩衝文件關聯着，不同的緩衝文件使用相同的 b: 變量是安全的，它們互不影響。

作用域前綴其實是個字典

以上介紹的各種作用域前綴，不僅是種語法約定的標記，它們本身也是個變量，是可以容納保存其他變量的字典類型變量。關於字典，在後續章節再詳述。這裏只能介紹幾個演示示例來體會一下這種特性。

爲了操作環境一致，也假設按上節的“裸裝”Vim 啓動：（不過其實不太影響，也不必太拘泥）

```
$ cd ~/.vim/vimllearn
$ vim -u NONE helloworld.txt
```

現在已用 vim 打開了一個 helloworld.txt 文件。在命令行輸入以下 ex 命令：

```
: let x = 1
```

```
: echo x
: echo g:x
: echo g:.x
: echo g:['x']
: echo g:
```

你可以每次按冒號進入命令行逐行輸入，也可以先進入 Ex 模式，連續輸入這幾行，效果是一樣的（以後不再註明）。

首先用“:let”命令定義了一個 x 變量，命令行語句默認是全局變量。然後用:echo 命令使用幾種不同寫法來引用讀取這個變量的值，這幾種寫法都是等效的。最後將 g: 當作一個整體變量打印顯示。它就是個全局字典，而裏面包含了 x 鍵，值就是 1。（如果按正常的 vim 啓動，你的 vimrc 以及各種插件可能會提供很多全局變量，那麼 echo g: 的內容可能很多，不只 x 喲）

然後我們再寫個腳本觀察下 s: 變量。‘:e hello2.vim’，輸入以下內容並保存：

```
1 " File: ~/.vim/vimllearn/hello2.vim
2 let s:hello = 1
3 let s:world = 2
4 let s:hello_world = s:hello + s:world
5 echo s:
```

腳本寫完了，在 ex 命令行輸入以下幾條測試下：

```
: source %
: echo s:
: echo s:hello
```

可見，‘:source %’命令能正常執行，腳本內的‘:echo s:’打印出了該腳本內定義的所有 s: 腳本變量。但在命令行直接試圖訪問 s: 變量則報錯了。

在腳本中也可以訪問全局變量。可以自行嘗試在 hello2.vim 中加入對剛纔在命令行定義的 g:x 變量的訪問。不過在實際的編程中，別在腳本中依賴在命令行建立的全局變量。

然後再測試下 b: 變量，直接在命令行執行以下語句吧：

```
: let b:x = 2
: echo b:x
: echo x
: echo g:x
: e #
: echo b:x
: echo b:
: echo x
```

```
: e #
: echo b:x
: echo b:
: echo x
```

這裏，‘:e #’ 表示切換編輯另一個文件。在實際工作中，或者用快捷鍵 <Ctrl-^> 更方便，不過在本教程中，爲說明方便，採用 ex 命令 ‘:e #’。在本例中，vim 啓動時打開的文件是 helloworld.txt，後來又編輯了 hello2.vim；此時用 ‘:e #’ 命令就切回編輯 helloworld.txt 了，再執行 ‘:e #’ 就再次回到 hello2.vim 中，這是輪換的效果。

這個示例結果表明，在編輯 hello2.vim 時定義了一個 b:x 變量，這與全局的 x 變量是互不衝突的。但是在換到編輯 helloworld.txt 時，b:x 變量就不存在了，因爲並未在該緩衝文件中定義 b:x 變量呀。重新回到編輯 hello2.vim 文件時，b:x 變量又能訪問了。這也說明當緩衝文件 “不可見” 時，vim 內部管理它的對象實體其實並未 “消失” 呢。而全局變量 g:x 或 x 是始終能訪問的。

最後要指出的是，局部作用域 l: 與參數作用域 a: 不能像 s: 或 b: 這樣當作整體的字典變量，是兩個例外。VimL 這樣處理的原因，可能一是沒必要，二是沒效率。函數體內的局部作用域與參數作用域太窄，沒必要將局部變量另外保存一個字典；而且有效時間太短，函數在棧上反覆重建銷燬，額外維護一個字典沒有明顯好處就不浪費了。另外若要表示所有函數參數另有一個語法變量 a:000 可實現其功能。

其他特殊變量前綴 $ v: &

這幾個符號其實並不是作用域標記。不過既然也是變量前綴，也就一道說明一下，也好加以區分。

含 $ 前綴的變量是環境變量。除了 vim 在啓動時會從父進程（如 shell）自動繼承一些環境變量，這些變量在使用上與全局變量沒什麼區別。不過要謹慎使用，一般建議只讀，不要隨便修改，沒必要的話也不要隨便創建多余的環境變量。（實際上環境變量與全局變量的最大區別是環境變量在 ex 命令中會自動展開爲當前的具體值，比如可直接使用 ‘:e $MYVIMRC’ 編輯啓動加載的 vimrc 文件。但在 VimL 腳本中將環境變量當作全局變量使用完全沒問題）

含 v: 前綴的變量是 vim 內部提供的預定義常量或變量。用戶不能增刪這類特殊變量，也不能修改其類型與含義。比如 v:true 與 v:false 分別用於表示邏輯值 “真” 與 “假”。Vim 所支持的這類 v: 變量往往隨着版本功能的增加而增加。從與時俱進的角度講，vim 腳本中鼓勵使用這類變量增加程序的可讀性，但若想兼容低版本，還是考慮慎用。要檢查當前 vim 版本是否支持某個 v: 變量，只要用 ‘:help’ 命令查閱一下即可。而且 v: 本身也是個字典集合變量，可用 ‘:echo v:’ 命令查看所有這類變量。

含 & 前綴的變量表示選項的值，相當於把選項變量化，以便於在 VimL 中編程。所支持的選項集，也是由 Vim 版本決定的，用戶當然無法定義與使用不存在的選項。這部分內容在後面講選項設置時再行討論。

1.4 * 自動加載脚本機制

前文已提及，vim 脚本主要用‘:source’命令加載，然而很多情況下又不需要手動執行該命令。只要將脚本放在特定的目錄下，vim 就有個機制能自動搜尋並加載。

Vim 插件搜索目錄

首先要知道有 &runtimepath（常簡寫爲 &rtp）這個選項。它與系統的環境變量 $PATH 有點類似，就是一組有序的目錄名稱，用於 Vim 在許多不同情況下搜尋 *.vim 脚本文件的。你可以在命令行輸入‘:echo &rtp’查看當前運行的 vim 有哪些“運行時目錄”，一般都會包含 ~/.vim 這個目錄。

• 除了 vim 啓動時的第一個配置文件 vimrc，運行時需要加載的脚本，一般都是從 &rtp 目錄列表中搜索的。

• vim 啓動時，會在所有 &rtp 目錄下的 plugin/ 搜索 *.vim 文件，並加載所有找到的脚本文件。需要注意的是在 plugin/ 子目錄下的所有脚本也會自動加載。除非你先在 vimrc 中用選項禁用加載插件這個行爲。

• 當一個文件類型 &filetype 被識別時，Vim 會從所有 &rtp 目錄下的 ftplugin/ 子目錄中搜索以文件類型開始的脚本文件，然後加載執行。比如編輯一個 cpp 文件時，ftplugin/ 目錄下的 cpp.vim cpp_*.vim cpp/*.vim 都會被加載。

所以，我們自己寫的脚本，如果想讓它在 vim 啓動時自動生效，就扔到 ~/.vim/plugin/ 目錄下，想只針對某種文件類型生效，就扔到 ~/.vim/ftplugin/ 目錄下。

目前主流的第三方插件，也會遵循這種子目錄規範，然後安裝時一般會將整個目錄添加到 &rpt 中，以便讓 Vim 找到對應的脚本。

VimL 的自動加載函數（延時加載）

Vim 一直有個追求的目標是啓動快。當插件越來越多時，vim 啓動時要解析大量的脚本文件，就會被拖慢了。這時就出現了一個 autoload 自動加載函數的機制，這個巧妙的方法可算是 VimL 發展的一個里程碑吧。而在這之前，須由用戶在 plugin/*.vim 的複雜脚本中用極具巧妙的編程技巧，纔好實現延時加載。

雖然還沒有講到 VimL 的函數，但也可以在這裏解釋自動加載函數的原理與過程，畢竟這不需要涉及到函數的具體實現。

例如，有一個 ~/.vim/autoload/foo.vim 脚本（或在其他任一個 &rtp 目錄下的 autoload/ 子目錄也行），該脚本內定義一個函數 foo#bar()，其中 # 之前的部分必須與脚本文件名 foo.vim 相同。將有以下故事發生：

• 在 vim 啓動時，完全不會讀取 foo.vim 文件，也不知道它裏面可能定義了什麼複雜的脚本內容。

• 當 foo#bar() 第一次被調用時，比如從命令行中執行‘:call foo#bar()’，vim 發現 foo#bar 這個函數未定義，就會試圖從這個函數名分析出它可能定義於 foo.vim 文件中。然

後就從 &rtp 目錄列表中，依次尋找其中 autoload/ 子目錄的 foo.vim 文件。將所找到的第一個 foo.vim 脚本加載，並停止繼續尋找。如果在所有 &rtp 目錄下都找不到，那就是個錯誤了。

• 加載（即:source）完 foo.vim ，再次響應 ':call foo#bar()' 的函數調用，就能正常執行了。

• 如果 foo.vim 文件中其實並沒有定義 foo#bar() 這個函數，比如手誤把函數名寫錯了，寫成了 foo#Bar()，則 vim 在二次嘗試執行 ':call foo#bar()' 時依然報錯說 "函數未定義"。

• 如果此後再次調用 ':call foo#bar()'，由於文件已加載，該函數是已定義的了，vim 就不需要再次尋找 foo.vim 文件了，直接執行就是。

• 如果 foo.vim 文件中還定義了一個 foo#bar2() 函數，由於之前是加載整個文件，foo#bar2() 也是個已定義函數，也就可以直接調用到 ':call foo#bar2()'。

• 如果嘗試調用一個 foo.vim 文件中根本不存在函數，如 ':call foo#nobar()'。即使之前已經加載過 foo.vim 一次，由於這個 foo#nobar 函數未定義，vim 會再次從 &rtp 目錄找到這個 foo.vim 文件再加載一次，然後再嘗試 ':call foo#nobar()' 依然出錯報錯。

各種細節過程可能很複雜，但總體思想還是很簡單，就是延時加載，只有在必要時才額外加載脚本。從用戶使用角度，只要注意幾點：

• 函數名 foo#bar() 必須與文件名 foo.vim 完全一致（大小寫也最好一致）。如果脚本是在 autoload 的深層子目錄下，那函數名也必須是相對於 autoload 的路徑名，把路徑分隔符 / 替換爲 # 就是。即在 autoload/path/to/foo.vim 文件中定義的函數名應該是 path#to#foo#bar()。

• 從使用便利性上，一般是會定義快捷鍵或命令來調用 # 函數，並在首次使用時觸發相關脚本的加載。

• # 函數是全局作用域的，也可以認爲各層 # 是完整的命名空間，當然從任何地方訪問時都須使用路徑全名，即使從相同的脚本內訪問也須用全名。

• 全局變量也可以用 # 命名，如 g:path#to#foo#varname 也能觸發相應脚本文件的自動（延時）加載，不過一般沒有函數應用那麼廣泛。

• 儘量將複雜業務邏輯代碼寫在 # 自動加載函數中，有時要注意不同 &rtp 目錄下同名文件的屏蔽效應。

利用 VimL 的這個自動加載機制，還有效地避免了全局變量（函數）名的衝突問題，因爲函數名包含了路徑名，而一般文件系統下是不會有重名文件的。唯一的問題是，這個函數名有點長。

第二章　VimL 語言基本語法

2.1　變量與類型

VimL 語言的變量規則與其他大多數語言一樣，可以（只允許）由字母、數字與下劃線組成，且不能以數字開頭。特殊之處在於還可以在變量名之前添加可選的作用域前綴，如“g: l: s: b: w: t:”（a: 又有點特殊，在定義函數參數時不要前綴，而在使用參數時需要前綴），這在第一章有專門討論，此不再敘說。

VimL 所支持的變量（值）類型可由幫助（‘:help type()’）查看。其中最主要最常用的有數字（number）、字符串（string）、列表（list）與字典（dictionary）四種，或者可以再進一步歸納爲三種，因爲前兩種（數字與字符串）在絕大數情況下自動轉換，在使用時幾乎不必考慮其類型差別，只須知道它表示“一個值”，所以也稱作標量。而列表與字典變量則是“多個值”的集合，所不同在於獲取其中某個值的索引方式不同。

標量：數字與字符串

數字是（number）直譯，其實就是其他大多數語言所稱的整數（int）。數字是有符號的，包括正負數與 0 ，其取值範圍與 Vim 的編譯版本有關。經筆者的測試，vim8.0 支持 8 字節的有符號整數，低版本只支持 4 字節的有符號整數。數字經常（或常規功能）是用於命令的地址參數表示行號，或命令的重複次數，一般情況下不必考慮數字溢界的問題。當你需要用到 VimL 來表達很大的整數時，纔要小心這個潛在的問題。

字符串也簡單，用單引號或雙引號括起來即可，它們的語義是完全一致的。不過有以下使用建議原則：

- 一般使用單引號表示字符串，如 string，畢竟雙引號還可用於行末註釋，儘量避免混淆。
- 如果需要使用轉義如 \n \t，則使用雙引號，單引號不支持轉義。
- 如果字符串包含一種引號，則使用另一種引號括起整個字符串。
- 如果有包含一層引用，則內外層用不同的引號。

數字變量支持常見的數學運算（加減乘除模，+-*/%），字符串只支持連接運算符，用點號 (.) 表示。此外，一些內建函數與命令也會要求其參數是數字或字符串。也就是數字與字符串有不同的使用環境，VimL 便能依據上下文環境將數字或字符串進行自動轉換，規則如下：

- 數字轉字符串表示是顯而易見的，就是十進制數的 10 個數字字符表示法；
- 字符串轉數字時，只截前面像數字的子串，若不以數字字符開頭，則轉爲數字 0 。

請測試：

```
: echo 'string' . 123
: echo '123'
: echo '123' + 1
: echo '123string' + 1
: echo '1.23string' + 1
: echo 'string123' + 1
```

需要特別注意的是字符串‘1.23’只會自動轉爲字數 1 ，而不是浮點數 1.23。

在 VimL 中輸入數字常量時，也支持按二進制、八進制、十六進制的表示法，不過自動轉字符串時只按十進制轉換。請自行觀察以下結果，如果不懂其他進制可無視。

```
: echo 0xff
: echo 020
: echo 0b10
: echo 0b10 + 3
: echo 'string' . 0xff
```

列表：有序集合

列表，在其他語言中也有的叫數組，就是許多值的有序集合。VimL 的列表有以下要點：

- 創建列表語法，中括號整體，逗號分隔元素：‘:let list = [0, 1, 2, 3]’
- 用中括號加數字索引訪問元素：‘:echo list[1]’
- 索引從 0 開始，支持負索引，-1 表示最後一個元素，訪問不存在索引時報錯。
- 索引也可以用整數變量。
- 不限長度，在需要時會自動擴展容量。

在列表創建後，用內建函數 add() 與 remove() 動態增刪元素，用 len() 函數取得列表長度（元素個數）。例如：

```
: let list = [0, 1, 2, 3]
: echo list
: call add(list, 4)
: call add(list, 5)
: echo list
: call remove(list, -1) | echo list
: call remove(list, 1) | echo list
```

字典：無序集合

字典，在其他語言中可能叫 Hash 表或散列表，就是許多“鍵-值”對的集合。與列表最大的不同在於，它不是用數字索引來訪問其內的元素，而是用字符串索引（鍵）來訪問元素。字典在內部存儲方式是無序的，但通過鍵訪問元素的速度極快。

定義與使用字典的語法示例如下：

```
: let dict = {'x': 1, 'y': 2, 'z': 3,}
: echo dict
: echo dict['x']
: echo dict.y
: let var = 'z'
: echo dict[var]
: let dict['u'] = 4
: let dict.v = 5
: echo dict
```

語法要點：

- 字典用大括號 {} 整體。
- 每個鍵值對用逗號分隔，鍵與值用冒號分隔，鍵一般有引號表字符串。
- 大括號內的空白是可選的，最後一個逗號也是可選的。
- 訪問字典內某個元素時，仍是中括號 [] 索引，鍵放在中括號中。
- 在創建字典或訪問元素時，鍵既可用引號引起的常量字符串，也可用字符串變量，數字變量自動轉換爲字符串。
- 當一個鍵是普通常量字符串（可用作變量名的字符串）時，可不用中括號加引號索引，而簡潔地用點號索引，二者等價。
- 不能訪問不存在的鍵，否則報錯。
- 能直接對不存在的鍵賦值，表示對字典增加一個鍵值對元素。

刪除變量

創建（或叫定義）變量用‘:let’命令，相應的也就有‘:unlet’命令用於刪除一個變量。一般情況下沒必要刪除一個標量，因爲它也佔不了多少內存，需要重定義時也可以重新賦值。但對於列表與字典，有時比較在意其集合意義，可以用‘:unlet’刪除其中一個值，加上對應的索引即可，如果不加索引，則表示刪除整個列表或字典。

例如：假設 list 與 dict 變量已如上定義：

```
: unlet list[1] | echo list
: unlet list[-1] | echo list
: unlet dict['u'] | echo dict
: unlet dict.v | echo dict
```

如果要刪除的變量或字典（列表）不存在的索引，‘:unlet’會報錯。如果想繞過該錯誤檢測，則可用‘:unlet!’命令。

在 vim8.0 版本之前，標量、列表、字典三者是不互通的。如果 list 已被定義成了一個列表變量，那麼它就不能用‘:let’重賦值爲一個字典或字符串或其他什麼，但允許重賦值爲另一個列表變量。如果一定要改變 list 的變量類型，只能先‘:unlet’它，再重新‘:let’它爲其他任意變量。

在 vim8.0 版本之後，不再有這個限制，不會再報諸如“類型不匹配”的錯誤了，更好地體現了動態弱類型的特點。然而，良好的命名規範要求變量名望文生義，在同一個範圍的同一個變量名，前後用之於表達完全不同類型的變量，並不是個好習慣。

浮點數

雖然浮點數在 VimL 中用的比較少，但畢竟還是支持的。

- 浮點數也叫小數，支持科學記數法。
- 數字（整數）可自動轉爲浮點數。
- 浮點不能自動轉爲整數，也不能自動轉爲字符串。
- 整數運算結果仍是整數，浮點數運算結果仍是浮點數。
- 浮點數取整後仍是浮點數，不是整數。

請看以下示例：

```
: echo 1.23e3
: let int = 123 | let float = 1.23 | let str = 'string'
: echo str . int
: echo str . float |"錯誤
: echo str . 5 / 3
: echo str . 5 % 3
: echo str . 5 / 3.0 |"錯誤
: echo str . 5 % 3.0 |"錯誤
: echo round(5/3.0)
: echo round(5/3.0) == 2
: echo round(5/3.0) . str |"錯誤
```

最後一行語句說明，雖然一個浮點數取整後看似與一個整數相等，但它仍然不是整數，所以不能與字符串自動連接。

要將一個字符串“顯式”轉換爲整數，可以與 0 相加；同理，要將整數“顯式”轉換爲字符串，可與空串“”相連接。VimL 還提供了另一個內建函數 string() 將任意其他類型轉換爲可打印字符串。於是，想將一個浮點數轉換爲“真正”的整數，可用如下操作：

```
: echo 0 + string(round(5/3.0))
: echo type(round(5/3.0))
```

```
: echo type(0 + string(round(5/3.0)))
```

注：行末註釋可用一個雙引號" 開始，但建議用 |" 更有適用性。| 表示分隔語句，只是後面一個語句是隻有註釋的空語句。

類型判斷

從 Vim8.0 開始，有一系列 vim 變量專門地用來表示各種變量（值）類型。比如 v:t_list 表示列表類型。如果要判斷一個變量是否爲列表類型，可用以下三種寫法中任何一種（但之前的低版本 Vim 只能用後兩種）：

```
if type(var) == v:t_list
if type(var) == 3
if type(var) == type([])
```

關於具有選擇分支功能的‘:if’語句，在下一節繼續講解。

2.2 選擇與比較

vim 在執行腳本時，一般是按順序逐條語句執行的。但如果只能像流水帳地順序執行，未免太無趣了，功能也弱爆了。本節介紹順序結構之外的最普遍的選擇分支結構，它可以根據某種條件有選擇地執行或不執行語句塊（一條或多條語句）。

在 VimL 中通過‘:if’命令來表示條件選擇，其基本語法結構是：

```
: if {expr}
:     " todo
: endif
```

如果滿足表達式 {expr}，或說其值爲“真”，則執行其後至‘:endif’之間的語句。貌似突然迸進了許多新概念，得先理一理。

表達式與語句

什麼叫表達式？這可難說了。我只能先描述下在 VimL 中什麼是與什麼不是表達式：

- 單獨的變量就是表達式，常量也是表達式，選項值（&option）也是，但選項本身不是；
- 函數調用是表達式；
- 表達式有值，表達式之間的合法運算的結果也還是表達式。
- 但表達式不是可執行語句，它只是語句的一部分。

至於語句，在第一章也講過。VimL 語句就是 Vim 的 ex 命令行。籠統地說，有時說到 ex 命令是指整個命令行，不過狹義地說，是指它第一個單詞所指代的關鍵命令。於是，VimL 語言的大部分語句，可認爲遵循以下範式：

```
VimL 語句 = ex 命令 + 表達式
```

爲什麼說大部分呢？因爲我們已經很熟悉的賦值語句如‘:let i=1’就不完全適合。在這裏，‘:let’是個命令，1 是個表達式。但 = 只是依附於‘:let’命令的特殊語義符號，它不是個表達式，也不是個運算符。變量 i 在被創建之前，也還算不上表達式。而 i=1 寫在一起，或爲了增加可讀性加些空白 i = 1，它也不是表達式，因爲它沒有值，（並不能像 C 語言那樣使用連等號賦值），下面這兩個語句是非法的：

```
: let i = j = 1
: let i = (j = 1)
```

在 VimL 中的常用語句中，除了這個基礎得有點平淡無奇的賦值語句，其他大多是“命令 + 表達式”範式的。比如已經大量使用的‘:echo’語句，以及上節介紹過的給列表添加一個元素的函數調用語句‘:call add(list, item)’。

然而，其實也不必太拘泥於這些概念名詞，理解就好。我們歸納出概念也不外是爲了更好地理解。

邏輯值與條件表達式

‘:if’命令（以及下一節要介紹的‘:while’命令）後面的表達式，就是一個條件表達式。它期望這個表達式的值的類型是邏輯值，即 type() 的結果是 v:t_bool(=6) 的值。如果值的類型不是邏輯值，則會自動將其他值轉換爲一個邏輯值。邏輯類型只有唯二的兩個值，v:true 表示真，v:false 表示假。

所以關鍵在於 VimL 如何判定其他值是否有真假，什麼是真，什麼是假？其轉換規則如何？這直接寫代碼測試一下吧：

```
: if 1 | echo v:true | endif
: if 0 | echo v:true | endif
: if -1 | echo v:true | endif

: if '0' | echo v:true | endif
: if '1' | echo v:true | endif
: if '' | echo v:true | endif
: if 'a' | echo v:true | endif

: if 1.23 | echo v:true | endif |" 非法用法
: if 0.23 | echo v:true | endif |" 非法用法
: if '0.23' | echo v:true | endif
: if '1.23' | echo v:true | endif
```

```
" 以下四條也都是非法用法
: if [1, 2, 3] | echo v:true | endif
: if [] | echo v:true | endif
: if {'x':1, 'y':2} | echo v:true | endif
: if {} | echo v:true | endif
```

注：由於語句比較簡單，就將‘:if’與‘:endif’直接寫在一行了，用 | 分隔子語句。正常代碼建議寫在不同行上且縮進佈局。

結果歸納於下：

- 數字 0 爲假，其他正數或負數爲真；
- 字符串先自動轉爲數字，轉爲 0 的話認爲假，能轉爲其他數字認爲真；
- 浮點數不能轉爲邏輯值，無法判斷真假；
- 列表與字典也不能直接判斷真假。

其實可進一步歸納爲一句話，在 VimL 中，只能對整數值判斷真假，0 是假的，其他都是真的，字符串先自動轉爲數字再判斷真假。其他類型的值不能直接判斷真假。(至 vim8.0 版本是此規則，後面是否會改就不得而知了)

然而，我們還是經常需要判斷其他類型的值的某種狀態。這時可以利用一個內建函數 empty() 來幫忙。它可以接收任何類型的一個參數，如果它是“空”的，就返回真 (v:true)，否則返回假 (v:false)。在很大程度上，它可以代替直接使用‘:if’的條件表達式，只不過在值上恰好是邏輯取反；優點則是寫法統一，適用於所有類型。

在上面這個例子中，可以都再次嘗試把‘:if’後面的表達式作爲 empty() 的參數執行看看結果，或用!empty() 取反判斷，如：

```
: if empty('0.23') | echo v:true | endif
: if !empty('a') | echo v:true | endif
```

綜合建議：用‘:if !empty(expr)’代替‘:if expr’，避免邏輯燒腦，並且大部分情況下應該是你想要的。

比較運算符

兩個整數進行相等性的比較，或大小性的比較，結果返回一個或真或假的邏輯值。整數支持的比較運算符包括：==, !=，>，>=，<，<=。

浮點數支持與整數相同的比較運算，但由於浮點誤差，不建議用相等性判斷。

字符串也支持與整數相同的那六個比較運算。雖然整數在直接‘:if’命令中自動轉爲數字處理，但在比較運算中表現良好，就是按正常的編碼序逐字符比較。不過有一點特別要注意的是，字符串比較結果受選項 &ignorecase 的影響，即有可能按忽略大小寫的方式來比較字符串。比如，觀察一下如下結果吧：

```
: set ignorecase
: echo 'abc' == 'ABC'
```

```
: set noignorecase
: echo 'abc' == 'ABC'
```

因此，爲了使比較結果不受用戶個人的 vimrc 配置 &ignorecase 的影響，VimL 另外提供兩套限定大小寫規則的比較運算符。在以上比較運算符之後再加 # 符號就表示強制按大小寫敏感方式比較，後面加上? 符號就表示強制按大小寫不敏感的方式比較。比如：

```
: echo 'abc' ==# 'ABC'
: echo 'abc' ==? 'ABC'
```

所以，強烈建議在進行字符串比較時，只用 ==# 或 ==? 系的比較運算符。當然由於弱類型，字符串變量與數字變量其實是不可分的，所以將 ==# 或 ==? 之類的運用於數字上比較，也是完全沒有關係的。

此外，字符串除了相等性比較，還有匹配性比較，即用 =~ !~ 運算符判斷一個字符串是否匹配另一個作爲正則表達式的字符串。正則表達式是另一個高級話題，這裏不再展開。當然，匹配運算符也有限定大小寫是否敏感的衍生運算符，而且一般建議用 =~# 與!~# 匹配，畢竟正則表達本身有表達大小寫的能力。

對於列表與字典變量，可進行相等性比較，但不能進行大小性比較。如果兩個列表或字典的對應元素都相等，則認爲它們相等。此外，列表與字典還另外有個同例性比較運算符，is 或 isnot。注意，這兩個是類似 == 的運算符號，不是關鍵詞，雖然它們用英文單詞來表示。同樣地，也有 is# 與 isnot? 的衍生運算，不過這主要爲了語法的統一整齊，其實 is#，is? 與 is 的結果是一致的。同例性比較的具體含義涉及實例引用的概念，這留待後面的章節繼續展開。

邏輯運算符

在 VimL 中的邏輯值所支持的或、且、非運算並無意外，分別用符號 || && ！表示就是，而且也支持短路計算特性。

• 或 expr1 || expr2 , 只要 expr1 或 expr2 其中一個是真，整個表達式就是真，兩個都是假纔是假。如果 expr1 已經是真的，expr2 不必計算就直接獲得真的結果。

• 且 expr1 && expr2，只有兩個表達式都是真，結果纔是真。如果 expr1 是假，則不必計算 expr2 就返回結果假。

• 非!expr，對錶達式真假取反。

if 分支流程

在瞭解這些邏輯值判斷之後，理解‘:if’的選擇分支語句就容易多了，其完整語法結構如下：

```
: if {expr}
:     {block_if}
```

```
: elseif {expr}
:     {block_elseif}
: elseif {expr}
:     {block_elseif}
: ......
: else
:     {block_else}
: endif
```

• 首先執行的是‘:if’後面的 expr 表達式，它可能只是個簡單表達式，也可能是多個邏輯值的複合運算，或者是很多表達式運算後得到的一個數字結果或邏輯值。只要它最終能被解釋爲真，就執行其後的 block_if 語句塊。

• 如果‘:if’的表達式爲假，則依次尋找下一個表達式爲真的‘:elseif’語句塊。

• 最後如果沒有真的‘:if’與‘:elseif’條件滿足，就執行‘:else’語句塊。

• 只有‘:if’與‘:endif’關鍵命令是必須的，‘:elseif’與‘:else’及其語句塊是可選的。

• 在任一條件下，最多隻有一個語句塊被執行，然後流程跳轉到‘:endif’之後，結束整個選擇分支流程。

• 如果沒有‘:else’語句塊，則在沒有任何一個條件滿足時，就不會執行任何一個語句塊。在有‘:else’時，則至少會執行一個語句塊。

注意：elseif 是直接將 else 與 if 這兩個單詞拼在一起的，中間沒有空格，也沒有縮寫。在許多不同的語言中，else if 的寫法可能是變化最多的。

在 VimL 中，目前也沒有 switch case 的類似語句，如果要實現多分支，只能疊加 elseif。

在非常簡單的 if else endif 語句中，也可以用條件表達式 expr1 : expr2 ? expr3，這類似於：

```
: if expr1
:     expr2
: else
:     expr3
: endif
```

整個表達式的值是 expr2 或 expr3 的值。至於條件表達式是否可以嵌套，這個我也不知道，反正我不用，也不建議用。就是條件表達式本身，也只推薦在一些有限的場合用，不推薦大量使用。因爲一開始以爲簡單的邏輯判斷，也可能以後會被修改得複雜起來，仍然是用‘:if’清晰一些。

然後推薦‘:if’的一個特殊技法。VimL 並沒有塊註釋，但是可以把多行語句嵌套放在‘:if 0 ... :endif’之間，然後其內的語句就完全不會被執行了，甚至有不合 VimL 語法的行也沒事。然而仍然只建議這樣“註釋”合法的語句行，因爲‘:if 0’的潛意識是在某個時刻可能需要將其改爲‘:if 1’以重新激活語句。這主要是用於更方便地切換測試某塊語句的運

行效果。

```
: if 0
:     這裏被注釋了
: endif
: echo 'done'
```

* 運算符優先級

本節爲講敘選擇分支語句，也引申講了不少有關語句、表達式、運算符的相關問題。落到實處就是各種運算符的使用了，這就需要特別注意運算符的優先級問題。在此並不打算羅列 VimL 的運算符優先級表，因爲到這裏可能還有些內容未覆蓋到。而且運算符優先級的問題太過瑣碎，只看一遍教程並無多大助益，需要經常查文檔，並自行驗證。可以通過這個命令‘:help expression-syntax’查看錶達式語法表，其中也基本是按運算符優先級從低到高排列的，請經常查閱。

雖然由於運算符優先級會引起一些自己意想不到的問題，但迴避這類問題的辦法也是很簡單的，這裏是一些建議：

- 首先按自己的理解去使用運算符，要相信大部分語言的設計都是人性化的，不會故意設些奇怪的違反常理的規則。
- 對於自己不確定優先級，或者發現運算結果不符合自己所想時，添加小括號組合，使表達式運算的次序明確化。
- 拆分複雜表達式，藉助中間變量，寫成多行語句，不要寫過長的語句。

2.3 循環與遍歷

程序比人體強大的另一個特性就是可以任勞任怨地重複地做些單調無聊（或有聊）的工作。本節介紹在 VimL 語言中，如何控制程序，命令其循環地按規則干活。

遍歷集合變量

首先介紹的是如何依次訪問列表如字典內的所有元素，畢竟在 2.1 節介紹的索引方法只適於偶爾訪問查看某個具體的元素。這裏要用到的是 for ... in 語法。例如遍歷列表：

```
: let list = [0, 1, 2, 3, 4,]
: for item in list
:     echo item
: endfor
```

在這個例子中，變量 item 每次獲取 list 列表中的一個元素，直到取完所有元素。相當於在循環中，依次執行‘:let item=list[0] :let item=list[1]’... 等語句。這個變量也可以叫做“循環變量”。遍歷列表保證是有序的。

對於字典的 for ... in 語法略有不同，因爲在字典內的每個元素是個鍵值對，不僅僅是值而已。其用法如下：

```
: let dict = {'x':1, 'y':2, 'z':3, 'u':4, 'v':5, 'w':6,}
: for [key,val] in items(dict)
:     echo key . '=>' . val
: endfor
```

注意：字典內的元素是無序的。

可以單獨遍歷鍵，利用內建函數 keys() 從字典變量中構造出一個列表：

```
: for key in keys(dict)
:     echo key . '=>' . dict[key]
: endfor
```

這裏的輸出結果應該與上例完全一致。

遍歷字典鍵時，如有需要，也可以先對鍵排個序：

```
: for key in sort(keys(dict))
:     echo key . '=>' . dict[key]
: endfor
```

遍歷字典還有個只遍歷值的方式，不過這種方式用途應該不多：

```
: for val in values(dict)
:     echo val
: endfor
```

總之，對於‘:for var in list’語句結構，var 變量每次獲取列表 list 內的一個值。字典不是列表，所以要利用函數 items() keys() values() 等先從中構造出一個臨時數組。

固定次數循環

如果要循環執行某個語句至某個固定次數，依然可利用 for ... in 語法。只不過要利用 range() 函數構造一個計次列表。例如，以下語句輸出 Hello World! 5 次：

```
: for _ in range(5)
:     echo 'Hello World!'
: endfor
```

這裏，我們用一個極簡的合變量，單下劃線 _ 來作爲循環變量，因爲我們在循環體中根本用不着這個變量。不過這種用法並不常見，這裏只說明可用 range() 實現計次循環。

那麼，range() 函數到底產生了怎樣的一個列表呢，這可用如下的示例來測試：

```
: for i in range(5)
```

```
:     echo i
: endfor
```

可見，range(n) 產出一個含 n 個元素的列表，元素內容即是數字從 0 開始直到 n，但不包含 n，用數學術語就叫做左閉右開。

其實，range() 函數不僅可以接收一個參數，還可以接收額外參數，不同個數的參數使得其產出意義相當不一樣，可用以下示例來理解一下：

```
: echo range(10)     |" => [0, 10)
: echo range(1,10)   |" => [1, 10]
: echo range(1,10,2) |" => 從1開始步長爲2的序列，不能超過 10
: echo range(0,10,2) |" => 從0開始步長爲2的序列，恰好包含 10
```

利用 range() 函數的這個性質，也就可以寫出不同需求的計次 for ... in 循環。

注：VimL 沒有類似 C 語言的三段式循環 for(初化; 條件; 更新)。只有這個 for ... in 循環，在某些語言中也叫 foreach 循環。

不定次數循環

不定循環用‘:while’語句實現，當條件滿足時，一直循環，基本結構如：

```
: let i = 0
: while i < 5
:     echo i
:     let i += 1
: endwhile
```

用‘:while’循環一個重要的注意點是須在循環前定義循環變量，並記得在循環體內更新循環變量。否則容易出現死循環，如果出現死循環，vim 沒響應，一般可用 Ctrl-C 中斷腳本或命令執行。

如果‘:while’條件在一開始就不滿足，則‘:while’循環一次也不執行。在‘:for ...in’循環中，空列表也是允許的，那就也不執行循環體。

在某些情況下，死循環是設計需求，那就可用‘:while 1’或‘:while v:true’來實現，而 for 循環無法實現，因爲構建一個無限大的列表是不現實的。

循環內控制

循環除了正常結束，還另外有兩個命令改變循環的執行流程：

- ‘:break’結束整個循環，流程跳轉到“:endfor”或“:endwhile”之後。
- ‘:continue’提前結束本次循環，開始下次循環，流程跳轉到循環開始，對於“:for”循環來說，循環變量將獲取下一個值，對於“:while”循環來說，會再次執行條件判斷。

• 這兩個命令一般要結合‘:if’條件語句使用，在特定條件下才改變流程，否則沒有太多實際意義。

舉些例子：

```
: for i in range(10)
:     if i >= 5
:         break
:     endif
:     echo i
: endfor
: echo 'done'
```

這裏只打印了前 5 個數，因爲當 i 變量到達 5 時，直接 break 了。

```
: for i in range(10)
:     if i % 2
:         continue
:     endif
:     echo i
: endfor
: echo 'done'
```

在這裏，i % 2 是求模運算，如果是奇數，余數爲 1 ，‘:if’條件滿足後由於‘:continue’直接開始下一次循環，‘:echo i’就被跳過，所以只會打印偶數。

在用‘:while’循環時，要慎重用‘:continue’，例如以下示例：

```
: let i = 0
: while i < 10
:     if i % 2
:         continue
:     endif
:     echo i
:     let i += 1
: endwhile
: echo 'done'
```

這原意是將上個打印偶數的‘:for’循環改爲‘:while’循環，但是好像陷入了死循環，先 <Ctrl-C> 中止再來分析原因。那原因就是‘:continue’語句跳過了‘:let i+=1’的循環變量更新語句，使它陷在同一個循環中再也出不來了。

所以，如果你的‘:while’是需要更新循環變量的，而且還用了‘:continue’，最好將更新語句放在所有‘:continue’之前。不過就這個例子而言，若作些修改後，還要同時修

改一些判斷邏輯，才能實現原有意圖。

* 循環變量作用域與生存期

對於‘:while’循環，循環變量是在循環體之外定義的，它的作用域無可厚非應與循環結構本身同級。但對於‘:for’循環，其循環變量是在循環頭語句定義的，(可見‘:let’並不是唯一定義或創建變量的命令，‘:for’也可以呢)，那麼在整個‘:for’結構結束之後，循環變量是否還存在，值是什麼呢?

```
: unlet! i
: for i in range(10)
:     echo i
: endfor
: echo 'done: ' . i
```

在這個例子中，爲避免之前創建的變量 i 的影響，先調用‘:unlet’刪了它，然後執行一個循環，在循環結束查看這個變量的值。可見在循環結束後，循環變量仍然存在，且其值是‘:for’列表中的最後一個元素。

那麼空循環又會怎樣呢?

```
: unlet! i
: for i in []
:     echo i
: endfor
: echo 'done: ' . i
```

這個示例執行到最後會報錯，提示變量不存在。所以循環變量 i 並未創建。因此准確地說，循環變量是在第一次進入循環時被賦值而創建的，而空循環就沒能執行到這步。

再看一下示例:

```
: unlet! i
: for i in range(10)
:     echo i
:     unlet i
: endfor
: echo 'done: ' . i
```

在這個例子中，只在循環體最後多加了一個語句，‘:unlet i’將循環變量刪除了。這種寫法在 Vim7 以前版本中很常見。因爲列表中是可以保存不同類型的其他變量的，甚至包括另一個列表或字典。因此在後續循環中，循環變量將可能被重新賦與完全不同類型的值，這在 Vim7 是一個“類型不匹配”的錯誤。所以在每次循環後將循環變量刪除，能避免這

個錯誤，使之適用性更廣。在 Vim8 之後，這種情況不再視爲錯誤，所以這個‘:unlet’語句不是必要。只是在這裏故意加回去，討論一下循環變量作用域與生存期的問題。

運行這個示例，可見在循環打印了 10 個數字後，最後那條語句報錯，變量 i 不存在。這也是可理解的，因爲這個變量在每次循環中反覆刪除重建。在第 10 次循環結束後，刪除了 i，但循環無法再進入第 11 次循環，也就 i 沒有再重建，所以之後 i 就不存在了。

這裏想說明的問題是，如果從安全性考慮，或對變量的作用域有潔癖的話，可以在循環體內‘:unlet’刪除循環變量。這樣可避免循環變量在循環結束後的誤用，尤其是循環中有‘:break’時，退出循環時那個循環變量的最後的值是很不直觀的，你最好不要依賴它去做什麼事情（除非是有意設計並考慮清楚了）。不過這有個顯然的代價是反覆刪除重建變量會消耗一些性能（別說 VimL 反正慢就不注重性能了，性能都是相對的）。

小結

VimL 只有兩種循環，for ... in 與 while。語義語法簡單明瞭，沒有其他太多變種需要記憶負擔，掌握起來其實應該不難。

2.4 函數定義與使用

函數是可重複調用的一段程序單元。在用程序解決一個比較大的功能時，知道如何拆分多個小功能，尤其是多次用到的輔助小功能，並將它們獨立爲一個個函數，是編程的基本素養吧。

VimL 函數語法

在 VimL 中定義函數的語法結構如下：（另參考‘:help :function’）

```
1 function[!] 函數名(參數列表) 附加屬性
2     函數體
3 endfunction
```

在其他地方調用函數時一般用‘:call’命令，這能觸發目標函數的函數體開始執行，以產生它所設計的功效。如果要接收函數的返回值，則不宜用‘:call’命令，可用‘:echo’觀察函數的返回結果，或者用‘:let’定義一個變量保存函數的返回結果。實際上，函數調用是一個表達式，任何需要表達式的地方，都可植入函數調用。例如：

```
1 call 函數名(參數)
2 echo 函數名(參數)
3 let 返回值 = 函數名(參數)
```

注：這裏爲了闡述方便，除了關鍵命令，直接用中文名字描述了。因而不是有效代碼，在每行的前面也就不加: 了。

函數名

函數名的命令規則，除了要遵循普通變量的命令規則外，還有條特殊規定。如果函數是在全局作用域，則只能以大寫字母開頭。

因爲 vim 內建的命令與函數都以小寫字母開始，而且隨着版本提升，增加新命令與函數也是司空見慣的事。所以爲了方便避免用戶自定義命令與函數的衝突，它規定了用戶定義命令與函數時必須以大寫字母開頭。從可操作 Vim 的角度，函數與命令在很大程度上是有些相似功能的。當然，如果將 VimL 視爲一種純粹的腳本語言，那函數也可以做些與 Vim 無關的事情。

習慣上，腳本中全局變量時會加 g: 前綴，但全局函數一般不加 g: 前綴。全局函數是期望用戶可以直接從命令行用‘:call’命令調用的，因而省略 g: 前綴是有意義的。當然更常見的是將函數調用再重映射爲自定義命令或快捷鍵。

除了接口需要定義在全局作用域的函數外，其他一些輔助與實現函數更適合定義爲腳本作用域的函數，即以 s: 前綴的函數，此時函數名不強制要求以大寫字母開頭。畢竟腳本作用域的函數，不可能與全局作用域的內建函數衝突了。

函數返回值

函數體內可以用‘:return’返回一個值，如果沒有‘:return’語句，在函數結束後默認返回 0。請看以下示例：

```
1 function! Foo()
2     echo 'I am in Foo()'
3 endfunction
4
5 let ret = Foo()
6 echo ret
```

你可以將這段代碼保存在一個.vim 腳本文件中，然後用‘:source’加載執行它。如果你也正在用 vim 讀該文檔，可以用 V 選擇所有代碼行再按 y 複製，然後在命令行執行‘:@”’，這是 Vim 的寄存器用法，這裏不准備展開詳述。如果你在用其他工具讀文檔，原則上也可以將代碼複製粘貼至 vim 的命令行中執行，但從外部程序複製內容至 vim 有時會有點麻煩，可能還涉及你的 vimrc 配置。因此還是複製保存爲.vim 文件再‘:source’比較通用。

這段示例代碼執行後，會顯示兩行，第一行輸出表示它進到了函數 Foo() 內執行了，第二行輸出表明它的默認返回值是 0。這個默認返回值的設定，可以想像爲錯誤碼，當函數正常結束時，返回 0 是很正常的事。

當然，根據函數的設計需求，可以顯式地返回任何表達式或值。例如：

```
1 function! Foo()
2     return range(10)
3 endfunction
```

```
4
5 let ret = Foo()
6 echo ret
```

執行此例將打印出一個列表，這個列表是由函數 Foo() 生成並返回的。

注意一個細節，這裏的‘:function!’命令必須加! 符號，因爲它正在重定義原來存在的 Foo() 函數。如果沒有! ，vim 會阻止你重定義覆蓋原有的函數，這也是一種保護機制吧。用戶加上! 後，就認爲用戶明白自己的行爲就是期望重定義同名函數。

一般在寫腳本時，在腳本內定義的函數，建議始終加上! 強制符號。因爲你在調試時可能經常要改一點代碼後重新加載腳本，若沒有! 覆蓋指令，則會出錯。然後在腳本調試完畢後，函數定義已定稿的情況下，假使由於什麼原因也重新加載了腳本，也不外是將函數重定義爲與原來一樣的函數而已，大部分情況下這不是問題。（最好是在正常使用腳本時，能避免腳本的重新加載，這需要一些技巧）

不過這需要注意的是，避免不同腳本定義相同的全局函數名。

函數參數

在函數定義時可以在參數表中加入若干參數，然後在調用時也須使用相同數量的參數：

```
1 function! Sum(x, y)
2     return a:x + a:y
3 endfunction
4
5 let x = 2
6 let y = 3
7 let ret = Sum(x, y)
8 echo ret
```

在本例中定義了一個簡單的求和函數，接收兩個參數；然後調用者也傳入兩個參數，運行結果毫無驚喜地得到了結果 5 。

這裏必須要指出的是，在函數體內使用參數 x 時，必須加上參數作用域前綴 a:，即用 a:x 纔是參數中的 x 形參變量。a:x 與函數之外的 x 變量（實則是 g:x）毫無關係，如果在函數內也創建了個 x 變量（實則是 l:x），a:x 與之也無關係，他們三者是互不衝突相擾的變量。

參數還有個特性，就是在函數體內是隻讀的，不能被重新賦值。其實由於函數傳參是按值傳遞的。比如在上例中，調用 Sum(x, y) 時，是把 g:x 與 g:y 的值分別拷貝給參數 a:x 與 a:y ，你即使能對 a:x，a:y 作修改，也不會影響外面的 g:x，g:y，函數調用結束後，這種修改毫無影響。然而，VimL 從語法上保證了參數不被修改，使形參始終保存着當前調用時實參的值，那是更加安全的做法。

爲了更好地理解參數作用域，改寫上面的代碼如下：

```
function! Sum(x, y)
    let x = 'not used x'
    let y = 'not used y'

    echo 'g:x = ' . g:x
    echo 'l:x = ' . l:x
    echo 'a:x = ' . a:x
    echo 'x = ' . x

    let l:sum = a:x + a:y
    return l:sum
endfunction

let x = 2
let y = 3
let ret = Sum(-2, -3)
echo ret
```

在這個例子中，調用函數 Sum() 時，不再傳入全局作用域的 x，y 了，另外傳入兩個常量，然後在函數體內查看各個作用域的 x 變量值。

結果表明，在函數體內，直接使用 x 代表的是 l:x，如果在函數內沒定義局部變量 x，則使用 x 是個錯誤，它也不會擴展到全局作用域去取 g:x 的值。如果要在函數內使用全局變量，必須指定 g: 前綴，同樣要使用參數也必須使用 a: 前綴。

雖然在函數體內默認的變量作用域就是 l:，但我還是建議在定義局部變量時顯式地寫上 l:，就如定義 l:sum 這般。雖然略顯麻煩，但語義更清晰，更像 VimL 的風格。函數定義一般寫在腳本文件，只用輸入一次，多寫兩字符不多的。

至於腳本作用域變量，讀者可自行將示例保存在文件中，然後也創建 s:x，s:y 變量試試。當然了，在正常的編程腳本中，請不要故意在不同作用域創建同名變量，以避免不必要的麻煩。(除非在某些特定情境下，按設計意圖有必要用同名變量，那也始終注意加上作用域前綴加以區分)

函數屬性：abort

VimL 在定義函數時，在參數表括號之後，還可以選擇指定幾個屬性。雖然在幫助文檔 ‘:help :function’ 中也稱之爲 argument，不過這與在調用時要傳入的參數是完全不同的東西。所以在這我稱之爲函數屬性。文檔中稱之爲 argument 是指它作爲 ‘:function’ 這個 ex 命令的參數，就像我們要定義的函數名、參數表也是這個命令的 “參數”。

至 Vim8.0 ，函數支持以下幾個特殊屬性:

- abort，中斷性，在函數體執行時，一旦發現錯誤，立即中斷運行。
- range，範圍性，函數可隱式地接收兩個行地址參數。
- dict，字典性，該函數必須通過字典鍵來調用。
- closure，閉包性，內嵌函數可作爲閉包。

其中後面兩個函數屬性涉及相對高深的話題，留待第五章的函數進階繼續討論。這裏先只討論前兩個屬性。

爲理解 abort 屬性，我們先來看一下，vim 在執行命令時，遇到錯誤會怎麼辦?

```
: echomsg 'before error'
: echomsg error
: echomsg 'after error'
```

在這個例子中，第二行是個錯誤，因爲 echo 要求表達式參數，但 error 這個詞是未定義變量。這裏用 echomsg 代替 echo 是因爲 echomsg 命令的輸出會保存在 vim 的消息區，此後可以用‘:message’命令重新查看；而 echo 只是臨時查看。

將這幾行語句寫入一個臨時腳本，比較 ~/.vim/vimllearn/cmd.vim ，然後用命令加載“:source ~/.vim/vimllearn/cmd.vim”。結果表明，雖然第二行報錯了，但第三行仍然執行了。

不過，如果在 vim 下查看該文檔，將這幾行復制到寄存器中，再用‘:@”’運行，第三行語句就似乎不能被執行到了。然而這不是主流用法，可先不管這個差異。

然後，我們將錯誤語句放在一個函數中，看看怎樣?

```
1 function! Foo()
2     echomsg 'before error'
3     echomsg error
4     echomsg 'after error'
5 endfunction
6
7 echomsg 'before call Foo()'
8 call Foo()
9 echomsg 'after call Foo()'
```

將這個示例保存在 ~/.vim/vimllearn/t_abort1.vim，然後‘:source’運行。結果錯誤之後的語句也都將繼續執行。

在函數定義行末加上 abort 參數，改爲:

```
: function! Foo() abort
```

重新‘:source’執行。結果表明，在函數體內錯誤之後的語句不再執行，但是調用這個出錯函數之後的語句仍然執行。

現在你應該明白 abort 這個函數屬性的意義了。一個良好的編程習慣是，始終在定義

函數時加上這個屬性。因爲一個函數我們期望它執行一件相對完整獨立的工作，如果中間出錯了，爲何還有必要繼續執行下去。立即終止這個函數，一方面便於跟蹤調試，另一方面避免在錯誤的狀態下繼續執行可能造成的數據損失。

那爲什麼 vim 的默認行爲是容忍錯誤呢？想想你的 vimrc，如果中間某行不慎出錯了，如果直接終止運行腳本，那你的初始配置可能加載很不全了。Vim 在最初提供函數功能，可能也只是作爲簡單的命令包裝重用，所以延續了這種默認行爲。但是當 VimL 的函數功能可以寫得越來越複雜時，爲了安全性與調試，立即終止的 abort 行爲就很有必要了。

如果你寫的某個函數，確實有必要利用容忍錯誤這個默認特性，當然你可以選擇不加 abort 這個屬性。不過最好還是重新想想你的函數設計，如果真有這需求，是否直接寫在腳本中而不要寫在函數中更合適些。

* 函數屬性：range

函數的 range 屬性，表明它很好地繼承了 Vim 風格，因爲很多命令之前都支持帶行地址（或數字）參數的。不過 range 隻影響一些特定功能的函數與函數使用方式，而在其他情況下，有沒有 range 屬性影響似乎都不大。

首先，只有在用 ‘:call Fun()’ 調用函數時，在 ‘:call’ 之前有行地址（也叫行範圍）參數時，Fun() 函數的 range 屬性纔有可能影響。

那麼，什麼又是行地址參數呢。舉個例子，你在 Vim 普通模式下按 V 進入選擇模式，選了幾行之後，按冒號:，然後輸入 call Fun()。你會發現，在選擇模式下按冒號進入 ex 命令行時，vim 會自動在命令行加上 ‘<,’>。所以你實際將要運行的命令是 “:‘<,’ >call Fun()”。‘< 與’ > 是兩個特殊的 mark 位置，分別表示最近選區的第一行與最後一行。你也可以手動輸入地址參數，比如 1,5call Fun() 或 1,$callFun()，其中 $ 是個特殊地址，表示最後一行，當前行用. 表示，還支持 + 與 - 表示相對當前行的相對地址。

總之，當用帶行地址參數的 ‘:{range}call’ 命令調用函數時，其含義是要在這些行範圍內調用一個函數。如果該函數恰好指定了 range 屬性，那麼就會隱式地額外傳兩個參數給這個函數，a:firstline 表示第一行，a:lastline 表示最後一行。

比如若用 ‘:1,5call Fun()’ 調用已指定 range 屬性的函數 Fun() ，那麼在 Fun() 函數體內就能直接使用 a:firstline 與 a:lastline 這兩個參數了，其值分別爲 1 與 5。如果用 ‘:‘<,’>call Fun()’ 調用，vim 也會自動從標記中計算出實際數字地址來傳給 a:firstline 與 a:lastline 參數。函數調用結束後，光標回到指定範圍的第 1 行，也就是 a:firstline 那行。

如果用 ‘:1,5call Fun()’ 調用時，Fun() 卻沒指定 range 屬性時。那又該怎辦，Fun() 函數內沒有 a:firstline 與 a:lastline 參數來接收地址啊？此時，vim 會採用另一種策略，在指定的行範圍內的每一行調一次目標函數。按這個實例，vim 會調用 5 次 Fun() 函數，每次調用時分別將當前光標置於 1 至 5 行，如此在 Fun() 函數內就可直接操作 “當前行” 了。整個調用結束後，光標停留在範圍內的最後一行。

函數的 range 屬性的工作原理就是這樣，然則它有什麼用呢？這個函數在操作 vim 中的當前 buffer 是極有用的。舉個例子：

```
1  " File: ~/.vim/vimllearn/frange.vim
2
3  function! NumberLine() abort
4      let l:sLine = getline('.')
5      let l:sLine = line('.') . ' ' . l:sLine
6      call setline('.', l:sLine)
7  endfunction
8
9  function! NumberLine2() abort range
10     for l:line in range(a:firstline, a:lastline)
11         let l:sLine = getline(l:line)
12         let l:sLine = l:line . ' ' . l:sLine
13         call setline(l:line, l:sLine)
14     endfor
15 endfunction
16
17 finish
18
19 測試行
20 測試行
21 測試行
22 測試行
23 測試行
```

在這個腳本中，定義了一個 NumberLine() 不帶 range 屬性的函數，與一個帶 range 屬性的 NumberLine2() 函數。它們的功能差不多，就是給當前 buffer 內的行編號，類似 set number 效果，只不過把行號寫在文本行之前。

這裏用到的幾個內建函數稍作解釋下，getline() 與 setline() 分別表示獲取與設定文本行，它們的第一個參數都是行號，當前行號用 “.” 表示。line(‘.’) 也表示獲取當前行號。

如果你正用 vim 編輯這個腳本，直接用 ‘:source %’ 加載腳本，然後將光標移到 finish 之後，選定幾行，按冒號進入命令行，調用 ‘:‘<,’>call NumberLine()’ 或 ‘:‘<,’>call NumberLine2()’ 看看效果。可用 u 撤銷修改。然後可將光標移到其他地方，手動輸入數字行號代替自動添加的 ‘<,’> 試試看。

最後，關於使用 range 屬性的幾點建議：

- 如果函數實現的功能，不涉及讀取或修改當前 buffer 的文本行，完全不用管 range 屬性。但在調用函數時，也請避免在 ‘:call’ 之前加行地址參數，那樣既無意義，還導致重複調用函數，影響效率。
- 如果函數功能就是要操作當前 buffer 的文本行，則根據自己的需求決定是否添加

range 屬性。有這屬性時，函數只調用一次，效率高些，但要自己編碼控制行號，略複雜些。

- 綜合建議就是，如果你懂 range 就用，不懂就不用。

* 函數命令

‘:function’ 命令不僅可用來（在腳本中）定義函數，也可以用來（在命令行中）查看函數，這個特性就如 ‘:command，:map’ 一樣的設計。

- ‘:function’ 不帶參數，列出所有當前 vim 會話已定義的函數（包括參數）。
- ‘:function {name}’ 帶一個函數名參數，必須是已定義的函數全名，則打印出該函數的定義。由此可見，vim 似乎通過函數名保存了一份函數定義代碼的拷貝。
- ‘:function /{pattern}’ 不需要全名，按正則表達式搜索函數，因爲不帶參數的 ‘:function’ 可能列出太多的函數，如此可用這個命令過濾一下，但是也只會打印函數頭，不包括函數體的實現代碼，即使只匹配了一個函數。
- ‘:function {name}()’ 請不要在命令行中使用這種方式，在函數名之後再加小括號，因爲這就是定義一個函數的語法！

* 函數定義 snip

在實際寫 vim 腳本中，函數應該是最常用的結構單元了。然後函數定義的細節還挺多，endfunction 這詞也有點長（腳本中不建議縮寫）。如果你用過 ultisnips 或其他類似的 snip 插件，則可考慮將常用函數定義的寫法歸納爲一個 snip。

作爲參考示例，我將 fs 定義爲寫 s: 函數的代碼片斷模板：

```
snippet fs "script local function" b
" $1:
function! s:${1:function_name}(${2}) abort "{{{
        ${3:" code}
endfunction "}}}
endsnippet
```

關於 ultisnips 這插件的用法，請參考ultisnips

小結

函數是構建複雜程序的基本單元，請一定要掌握。函數必須先定義，再調用，通過參數與返回值與調用者交互。本節只講了 VimL 函數的基礎部分，函數的進階用法後面另有章節專門討論。

2.5 * 异常處理

異常是編程中相對高級的話題，也是比較有🄴議的話題。本教程旨在 VimL ，不可能展開去討論異常機制。所以如果你不瞭解異常，也不用異常，那就可完全跳過這節了。如果你瞭解異常，並且不反對用異常，那麼這裏只是告訴你，VimL 也提供了語法支持，可以讓你在腳本中使用異常，其基本語法結構如下：

```
try
    嘗試語句塊
catch /正則1/
    异常處理1
catch /正則2/
    异常處理2
...
finally
    收尾語句塊
endtry
```

大致流程是這樣的：先執行 try 下面的嘗試語句塊，如果這過程中不出現錯誤，那就沒 catch 什麼事了，但是如果有 finally，其後的收尾語句塊也會執行。麻煩在於如果嘗試語句塊中有錯誤發生，就會拋出一個錯誤。錯誤用字符串消息的形式，所以 catch 用正則表達式捕獲。由於錯誤消息可能有本地化翻譯，所以匹配錯誤號比較通用。如果 catch 沒有參數，則捕獲所有錯誤。一旦錯誤被某個 catch 正確匹配了，就執行其後的異常處理語句塊，然後如果有 finally 的話，收尾語句塊也會執行。

如果在 try 中出現了錯誤，既沒有 catch 捕獲，也沒有 finally 善後，那它就向上層繼續拋出這個錯誤。直到有地方處理了這個錯誤，如果一直沒能處理該錯誤，就終止腳本運行。

除了 vim 執行腳本中自動檢測錯誤拋出外，也有個命令‘:throw’可手動拋出。比較常見的在 catch 的異常處理塊中，只處理了部分工作後，用‘:throw’重新拋出錯誤讓後續機制繼續處理。‘:throw’不帶參數時重新拋出最近相同的錯誤，否則可帶上參數拋出指定錯誤。

雖然 VimL 也提供了這個一套完整的異常處理機制，但一般情況下用得不多。大約有以下原因：

- 使用 VimL 希望簡單，用上異常就似乎很複雜了。
- vim 腳本本身就很安全，只能在 vim 環境下運行，似乎干不了什麼壞事。而且 vim 早就有備份相關的配置，對編輯保存的文件都可以備份的。

所以，除非要寫個比較大與複雜的插件，用異常可能在代碼組織上更爲簡潔，提供更良好的用戶接口。

第三章　Vim 常用命令

在第二章已經介紹了 VimL 語言的基本語法，理論上來說，就可以據此寫出讓 vim 解釋執行的合法腳本了。然而，能寫什麼腳本呢？除了打印 “Hello World!”，以及高級點的用循環計算諸如 “1+2+...+100” 這樣人家好像也能心算的題目外，還能干嘛呢？

所以，如果要讓 vim 腳本真正有實用價值，還得掌握 vim 提供的內置命令，用以控制 Vim 或定製 Vim。本章就來介紹一些主要的、常用的命令。

Vim 是個極高自由度的文本編輯軟件，它在以下幾個層級上給用戶提供了自由度：

1. option 選項。預設了一個很龐大的選項集，用戶可以按自己的喜好設置每個選項的值（當然很多選項也可以接受默認值而假裝當它們不存在），這可能改變 Vim 的很多基礎表現與行爲。

2. map （快捷鍵）映射。一個簡單但非常強大的機制。用戶可以根據自己的習慣來重新映射各種模式下不同按鍵（及按鍵序列）的解釋意義。初入門的 Vimer 很容易沉迷於折騰各種快捷鍵。

3. command 自定義命令。Vim 是基於 ex 命令的，然後允許又你自定義 Ex 命令。可見這是比簡單映射更靈活強大的利器，當然它的使用要求也比映射要高一些。

4. VimL 腳本。進一步將命令升級爲腳本語言，據此開發插件，使得 Vim 的擴展性具有無限可能。在 Vim 社區已經涌現了很多優秀插件，大多可以直接拿來用。當自己掌握了 VimL 語言後，也就可以自己寫些插件來滿足自己的特殊需求或癖好。

本教程雖是旨在 VimL 腳本語言，但還是有必要從簡單的選項說起吧。

3.1　選項設置

選項分類與設置命令

設置選項的命令是 set。根據選項值的不同情況，可以將選項分爲以下三類：

1. 不需要值的選項，或者說是 bool 型的開關切換狀態的選項。這種選項有兩個相對立的選項名，分別用命令‘:set option’表示開啓選項，‘:set nooption’表示關閉選項。例如‘:set number’是設置顯示行號，‘:set nonumber’是設置不顯示行號。

2. 選項有一個值。用命令‘:set option=value’設定該類選項的值。選項值可以是數字或字符串，但字符串的值也不能加引號，就按字面字符串理解。也就是說，‘:set’後面

的參數，不是 VimL 的表達式，與‘:let’命令有根本的不同。這個命令更像是 shell 設置變量的語法，= 前後也最好不要用空格。

3. 選項允許有多個值，值之間用逗號分隔。設置命令形如‘:set option=val1,val2’。此外還支持 += 增量與 -= 減量語法，如‘:set option+=val3’或‘:set option-=val2’，表示在原來的“值集合”的基礎上增加某個值或移除某個值。

選項值變量

在選項名前面加個 & 符號，就將一個選項變成了相應的選項值變量。例如，以下兩條命令是等效的：

```
: set option=value
: let &option = value
```

與普通變量賦值一樣，= 前後的空格是可選的，這裏的空格只是一種編程習慣，爲增加可讀性。另外有以下幾點要注意：

1. 第一類選項，在用‘:set’命令時不需要等號，但是用‘:let &’命令時也要用等號將其值賦爲 1 或 0，分別表示開啓選項與關閉選項。同時 & 只允許作用在沒有 no 前綴的選項之前。比如‘:let &nonumber = 1’是非法的，只能用‘:let &number = 0’表示相同意圖。

2. 第二類選項，如果值是字符串，用‘:let &’命令時要將值用引號括起來，也就像普通變量賦值一樣，要求等號後面是合法的表達式。

3. 第三類選項，它的值也是一個由逗號分隔的（長）字符串，比如‘:echo &rtp’。並不能由於這類選項支持多個值就將 VimL 的列表賦給它，不過很容易通過 split() 函數從這類選項值中分隔出一個列表。

備註：選項設置‘:set’應是歷史淵源最早的命令之一吧。而‘:let’是後來 VimL 語言發展豐富起來提供的命令。兩者有不一樣的語法，所以又提供了這種等價轉換方法。

vimrc 配置全局選項

嚴格地說，‘:set’是設置全局選項的命令。既是影響全局的選項，一般是要第一時間在 vimrc 中配置的。最重要的是以下兩條配置：

```
: set nocompatible
: filetype plugin indent on
```

第一條配置是說不要兼容 vi，否則可能有很多 vim 的高級功能用不了。第二條配置（雖然不是 set 選項）是用 Vim 編寫程序源代碼必要的，意思是自動檢測文件類型，加載插件，自動縮進的意思。除非在很老舊的機器上，或爲了研究需要，一般都沒理由不加上這兩條至關重要的配置。

下面再介紹一些比較重要的幾類配置選項，當然這遠遠不夠全面。查看選項的幫助命令是‘:help options’，查看某一個選項的幫助是在用單引號括起選項名作爲幫助參數，例如‘:help ‘option’’。查看某個選項的當前值是命令‘:set option?’或‘:echo &option’。

- 編碼相關：encoding fileencodings fileencoding
  - encoding 是 Vim 內部使用的編碼。建議‘:set encoding=utf-8’。
  - fileencodings 是打開文件時，Vim 用於猜測檢測文件編碼的一個編碼列表。對中文用戶，建議‘:set fileencodings=ucs-bom,utf-8,gb18030,cp936,latin1’。
  - fileencoding （局部選項），當前文件的編碼，如果與 encoding 不同，在寫入時自動轉碼。用戶一般不必手動設這個選項，除非你想用另外一種編碼保存文件。
- 外觀相關：number/relativenumber wrap statusline/tabline
  - number 是在窗口左側加一列區域顯示行號，relativenumber 顯示相對行號，即相對光標所在的行的行號，當前行是 0，上面的行是負數，下面的行是正數。
  - wrap 是指很長的文本行折行顯示。一般良好風格的程序源文件不應出現長行，但 Vim 作爲通用文件編輯器，不一定只用於編輯程序。
  - statusline 是定製狀態欄，格式比較複雜，建議查看文檔，也有些插件提供了很炫酷的狀態欄。tabline 的定製格式與狀態欄一樣，在開多個標籤頁時才生效。
  - laststatus 什麼時候顯示狀態欄，建議用值 2 表示始終顯示狀態欄。
  - cmdheight 命令行的高度，默認只有 1 行太少，當命令行有輸出時可能經常要多按一個回車纔回到普通模式。建議 2 行，更多就浪費空間了。
  - wildmenu 這是在編輯命令行時，按補全鍵後，會臨時在狀態欄位置顯示補全提示。
- GUI 外觀：只在 gVim 或有 GUI 版本的 Vim 有效
  - guioptions 設置 GUI 各部件（菜單工具欄滾動條等）是否顯示。
  - clipboard 設置剪切板與 Vim 的哪個寄存器關聯。
- 顏色主題：
  - colorscheme 這是個單獨的命令，不是 set 選項。選擇一個顏色主題。顏色主題是放在運行時各路徑的 colors/ 子目錄的 *.vim 文件。
  - background 背景是深色 dark 或淺色 light。有的 colorscheme 只適於深色或淺色背景，有的則分別爲不同背景色定義不同的顏色主題。
  - term 與 t_Co 有的顏色主題可能還與終端與終端色數量有關。
  - cursorline 與 cursorcolumn 用不同格式高亮當前行與當前列，具體高亮格式由顏色主題定義。個人建議只高亮 cursorline 。
  - hlsearch 高亮搜索結果。
- 格式控制：
  - formatoptions 控制自動格式化文本的許多選項，建議看文檔。
  - textwidth 文本行寬度，超過該寬度（默認 78）時自動加回車換回。在編輯程序源文件時可用上個 formatoptions 選項控制只在註釋中自動換行但代碼行不自動換行。
  - autoindent smartindent 插入模式下回車自動縮進。
  - shiftwidth 縮進寬度。

◦ tabstop softtabstop 製表符寬度，軟製表符是行首按制符縮進的寬度。一般建議硬製表符寬度 tabstop 保持 8 不變，用 shiftwidth softtabstop 表示縮進。

◦ expandtab 插入製表符自動轉爲合適數量的空格。

◦ paste 將 Vim 的插入模式置於 “粘貼” 模式，從外部複製文本進 Vim 開啓該選項可避免一些副作用。但只建議臨時開啓該選項。

• 路徑相關:

◦ runtimepath 運行時路徑，簡稱（rtp）。vim 在運行時搜索腳本的一組路徑。一般不手動設置該值，如果有插件管理器管理插件的話。插件必須放在某個 &rtp 路徑下，現在流行的是將插件工程主目錄添加至 Vim 的 &rtp 中。

◦ packpath （Vim8 開始才支持）動態加載插件的搜索路徑，默認是 /.vim/pack。插件主目錄可置於 {packpath}/{packname}/opt/{plugin}。然後用 ‘:packadd’ 啓用插件。

◦ path 這是 vim 在尋找編輯文件，如 gf :find 等命令時所要搜索的一組目錄。

◦ tags 這是 vim 按標籤跳轉 Ctrl-] 或‘:tag’等命令所依據的標籤文件，默認是./tags,tags（相對路徑）。一般不建議修改默認值，但可以默認值基礎上添加更多的標籤文件，比如編輯一個工程時，將工程主目錄下的 tags 文件也加進來。

◦ autochdir 將當前路徑自動切換到當前編輯的文件所在的目錄。當你依賴一些管理工程類的插件時，可能要求當前路徑鎖定在工程主目錄，不宜開啓該選項。但是個人喜歡開啓這選項，這樣在用 ‘:e’ 命令打開同目錄下的其他文件時很方便。

Vim 所支持的選項實在是太多了。初學者建議參考前人經驗成熟的配置，用 ‘:help’ 查看每個選項的具體含義，然後決定這種選項是否適合自己。另外注意有些選項可能要配合起來才能發揮更好的效果。

VimL 控制局部選項

局部選項是隻影響當前緩衝文件或窗口（buffer/window）的選項。嚴格來說是局部選項值，而不是有另外一類選項。默認情況下每個新文件或窗口都繼承選項的全局值，但對於一些選項，可以爲該文件或窗口設定一個不同與全局的局部值。然而並不是所有選項都有局部值意義，在每個選項的幫助文檔中，會指明該選項是全局（global）或局部的（local to buffer 或 local to window）。

設置局部選項（值）用 ‘:setlocal’ 命令。如果目標選項沒有局部值，則等效 ‘:set’ 命令設置全局值。但是最好不要混用，避免誤解。局部選項值變量用 &l:option 表示。比如 number 行號就是個局部選項:

```
: set nonumber
: setlocal number
: echo &number
: echo &l:number
: echo &g:number
```

你可以將 vim 分裂出兩個窗口（':split' 或 ':vsplit'），在其中一個窗口上執行以上語句，試試看結果。需要注意的是，雖然局部選項值借用了變量的局部作用域前綴 l:，但它的默認規則又有點不同。看這裏的 &number 是默認的 &l:number 而不是 &g:number。事實上，普通的局部變量 l:var 根本不能在函數外的命令行使用。

當用 VimL 寫腳本時，如果要改變選項設置，且該選項支持局部值，最好用 ':setlocal' 只改變局部值。這也是編程的一大原則，儘量將影響局部化。下面介紹一些比較重要的局部選項設置：

- 文件類型：filetype
  - 大部分情況下，這個選項不用手動設置，也不用腳本顯式設置，打開自動檢測就可以自動根據後綴名設置相應的文件類型。不過在創建新的文件類型時，可能需要自己設置這個選項。
  - 文件類型插件，如 ~/.vim/ftplugin/*.vim 腳本內若涉及選項更改，也儘量用 ':setlocal' 只設局部選項。
- 緩衝類型：buftype
  - "buffer type" 與 "file type" 是兩個不同的概念。緩衝類型更加抽象，是 vim 內部用於管理緩衝的一些控制屬性，而文件類型是着眼於文件內容性質的。
  - buftype 的兩個重要的選項值是 nofile 與 nowrite，表示特殊的不用寫文件的 buffer，而這兩者又還有細微差別，具體請讀文檔。
- 其他 buffer 屬性：
  - buflisted 是否將當前緩衝記錄在緩衝列表中。
  - bufhidden 當緩衝不再任一窗口展示，如何處理該緩衝，有幾種不同的選項值。
  - modifiable 當前緩衝是否可修改，包括更改編碼與換行符格式也算種修改。

由於在 Vim 中，最主要的可見（可編輯）對象就只是 buffer，所以在一些複雜而細緻的插件中，經常會開闢一個輔助窗口，僅爲展示輔助內容，這就往往要設置一個特殊的 buftype 及其他一些 buffer 屬性。

此外，在腳本中，可能有需求只臨時改變某個選項值，處理完畢後再恢復原選項設置，這就要借且選項值變量了。處理流程大致如下：

```
: let l:save_option = &l:option
: let &l:option = ? |" 或者 setlocal option = ?
: " do something
: let &l:option = l:save_option
```

3.2 快捷鍵重映射

幾乎每個初窺門徑的 vimer 都曾爲它的鍵映射欣喜若狂吧，因爲它定製起來實在是太簡潔了，卻又似能搞出無盡的花樣。

快捷鍵，或稱映射，在 Vim 文檔中的術語叫 map，它的基本用法如下：

```
map {lhs} {rhs}
map 快捷鍵 鍵序列
```

其中快捷鍵 {lhs} 不一定是單鍵，也可能是一個（較短的）按鍵序列，然後 vim 將其解釋爲另一個（可能較長較複雜的）的按鍵序列 {rhs}。爲方便敘述，我們將 {lhs} 稱爲 “左參數”，而將 {rhs} 稱爲 “右參數”。左參數是源序列，也可叫被映射鍵，右參數是目標序列，也可叫映射鍵。

例如，在 vim 的默認解釋下，普通模式下大寫的 Y 與兩個小寫的 yy 是完全相同的功能，就是複製當前行。如果你覺得這浪費了快捷鍵資源，可將 Y 重定義爲複製當前行從當前光標列到列尾的部分，用下面這個映射命令就能實現：

```
: map Y y$
```

然而，映射雖然初看起來簡單，其中涉及的門道還是很曲折的。讓我們先回顧一下 Vim 的模式。

Vim 的主要模式

模式是 Vim 與其他大多數編輯器的一個顯著區別。在不同的模式下，vim 對用戶按鍵的響應意義有根本的差別。Vim 支持很多種模式，但最主要的模式是以下幾種：

• 普通模式，這是 Vim 的默認模式，在其他大多模式下按 <Esc> 鍵都將回到普通模式。在該模式下按鍵被解釋爲普通命令，用以完成快速移動、查找、複製粘貼等操作。

• 插入模式，類似其他 “正常” 編輯的模式，鍵盤上的字母、數字、標點等可見符號當作直接的字符插入到當前緩衝文件中。從普通模式進入插件模式的命令有：aAiIoO

  ◦ a 在當前光標後面開始插入，

  ◦ i 在當前光標之前開始插入，

  ◦ A 在當前行末尾開始插入，

  ◦ I 在當前行行首開始插入，

  ◦ o 在當前行下面打開新的一行開始插入，

  ◦ O 在當前行上面打開新的一行開始插入。

• 可視模式（visual），非正式場合下也可稱之爲 “選擇” 模式。在該模式下原來的移動命令變成改變選區。選區文本往往有不同的高亮模式，使用戶更清楚地看到後續命令將要操作的目標文本區域。從普通模式下，有三個鍵分別進入三種不同的可視模式：

  ◦ v（小寫 v）字符可視模式，可以按字符選擇文本，

  ◦ V（大寫 V）行可視模式，按行選擇文本（jk 有效，hl 無效），

  ◦ Ctrl-v 列塊可視模式，可選擇不同行的相同一列或幾列。（Vim 還另有一種 “select” 模式，與可視模式的選擇意義不同，按鍵輸入直接覆蓋替換所選擇的文本）

• 命令行模式。就是在普通模式時按冒號 ‘:’ 進入的模式，此時 Vim 窗口最後一行將變成可編輯輸入的命令行（獨立於當前所編輯的緩衝文件），按回車執行該命令行後回到普通

模式。本教程所說的 VimL 語言其實不外也是可以在命令行中輸入的語句。此外還有一種“Ex 模式”，與命令行模式類似，不過在回車執行完後仍停留在該模式，可繼續輸入執行命令，不必每次再輸入冒號。在“Ex 模式”下用‘:vi’命令纔回到普通模式。

大部分初、中級 Vim 用戶只要掌握這四種模式就可以了。對應不同模式，就有不同的映射命令，表示所定義的快捷鍵只能用於相應的模式下：

- 普通模式：nmap
- 插入模式：imap
- 可視模式：vmap （三種不同可視模式並不區分，也包括選擇模式）
- 命令模式：cmap

如果不指定模式，直接的 map 命令則同時可作用於普通模式與可視選擇模式以及命令後綴模式（Operator-pending，後文單獨講）。而 map! 則同時作用於插入模式與命令行模式，即相當於 imap 與 cmap 的綜合體。其實 vmap 也是 xmap（可視模式）與 smap （選擇模式）的綜合體，只是 smap 用得很少，vmap 更便於記憶（v 命令進入可視模式），因此我在定義可視選擇模式下的快捷鍵時傾向於用 vmap。

在其他情況下，建議用對應模式的映射命令，也就是將模式簡名作爲 map 的限定前綴。而不建議用太過寬泛的 map 或 map! 命令。

特殊鍵表示

在 map 系列命令中，{lhs} 與 {rhs} 部分可直接表示一般字符，但若要映射（或被映射）的是不可打印字符，則需要特殊的標記（<> 尖括號內不分大小寫）：

- 空格：<Space> 。映射命令之後的各個參數要用空格分開，所以若正是要重定義空格鍵意義，就得用 <Space> 表示。同時映射命令儘量避免尾部空格，因爲有些映射會把尾部空格當作最後一個參數的一部分。始終用 <Space> 是安全可靠的。
- 豎線：<BAR>。| 在命令行中一般用於分隔多條語句，因此要重定義這個鍵要用 <BAR> 表示。
- 歎號：<Bang>。! 可用於很多命令之後，用以修飾該命令，使之做一些相關但不同的工作，相當於特殊的額外參數。映射中要用到這個符號最好也以 <Bang> 表示。
- 製表符：<Tab>、回車：<CR>
- 退格：<BS>、刪除鍵：<DEL>、插入鍵：<Ins>
- 方向鍵：<UP> <DOWN> <LEFT> <RIGHT>
- 功能鍵：<F1> <F2> <F3> <F4> <F5> 等
- Ctrl 修飾鍵：<C-x> （這表示同時按下 Ctrl 鍵與 x 鍵）
- Shift 修飾鍵：<S-A>，對於一般字母，直接用大寫字母表示即可，如 A 即可，不必有 <S-a>。一般對特殊鍵可雙修飾鍵時纔用到，如 <C-S-a>。
- Alt <A-> 或 Meta <M-> 修飾鍵。在 term 中運行的 vim 可能不方便映射這個修飾鍵。
- 小於號：<lt>、大於號 <gt>

• 直接用字符編碼表示：<Char->，後面可接十進制或十六進制或八進制數字。如 <Char-0x7f> 表示編碼爲 127 那個字符。這種方法雖然統一，但如有可能，優先使用上述意義明確方便識記的特殊鍵名錶示法。

此外，還有幾個特殊標記並不是特指哪個可從鍵盤輸入的按鍵：

• <Leader> 代表 mapleader 這個變量的值，一般叫做快捷鍵前綴，默認是 \。同時還有個 <LocalLeader>，它取的是 maplocalleader 的變量值，常用於局部映射。

• <SID> 當映射命令用於腳本文件中（應該經常是這種情況），<SID> 用於指代當前腳本作用域的函數，故一般用於 {rhs} 部分。當 vim 執行映射命令時，實際會把 <SID> 替換爲 <SNR>dd_ 樣式，其中 dd 表示當前腳本編號，可用‘:scriptnames’查看所有已加載的腳本，同時也列出每個腳本的編號。

• <Plug> 一種特殊標記，可以避免與用戶能從鍵盤輸入的任何按鍵衝突。常用於插件中，表示該映射來自某插件。與 <SID> 關聯某一特定腳本不同，<Plug> 並不關聯特定插件的腳本文件。它的意義請繼續看下一節。

鍵映射鏈的用途與陷阱

鍵映射是可傳遞的，例如若有以下映射命令：

```
: map x y
: map y z
```

當用戶按下 x，vim 首先將其解釋爲相當於按下 y，然後發現 y 也被映射了，於是最終解釋爲相當於按下 z。

這就是鍵映射的傳遞鏈特性。那這有什麼用呢，爲什麼不直接定義爲‘:map x z’呢？假如 z 是個很複雜的按鍵命令，比如 LongZZZZZZZ，那麼就可先爲它定義一個簡短的映射名，如 y：

```
: map y LongZZZZZZZ
: map x1 y
: map x2 y
```

然後再可以將其他多個鍵如 x1 與 x2 都映射爲 y，不必重複多次寫 LongZZZZZZZ 了。然而，這似乎仍然很無趣，真正有意義的是用於 <Plug>。

假設在某個插件文件中有如下映射命令：

```
: map <Plug>(do_some_thing) :call <SID>AFunc()<CR>
: map x <Plug>(do_some_thing)
: map <C-x> <Plug>(do_some_thing)
: map <Leader>x <Plug>(do_some_thing)
```

在第一個映射命令中，其 {lhs} 部分是 <Plug>(do_some_thing)，這也是一個“按鍵序列”，不過第一鍵是 <Plug>（其實不可能從鍵盤輸入的鍵），然後接一個左括號，接着

是一串普通字符按鍵，最後還是個右括號。其中左右括號不是必須的，甚至可以不必配對，中間也不一定只能普通字符，加一些任意特殊字符也是允許的。不過當前許多優秀的插件作者都自覺遵守這個範式：<Plug>(mapping_name)。

該命令的 {rhs} 部分是 ':call <SID>AFunc()<CR>'，表示調用當前腳本中定義的一個函數，用以完成實際的工作。然而 <Plug>... 是不可能由用戶按出來的鍵序列，所以需要再定義一個映射 ':map x <Plug>...'，讓一個可以方便按出的鍵 x 來觸發這個特殊鍵序列 <Plug>...，並最終調用函數工作。當然了，在普通模式下的幾乎每個普通字母 vim 都有特殊意義（不一定是 x，而 x 表示刪除一個字符），你可能不應該重定義這個字母按鍵，可加上 <Leader> 前綴修飾或其他修飾鍵。

那麼爲何不直接定義 ':map x :call <SID>AFunc()<CR>' 呢？一是爲了封裝隱藏實現，二是可爲映射取個易記的映射名如 <Plug>(mapping_name)。這樣，插件作者只將 <Plug>(mapping_name) 暴露給用戶，用戶也可以自己按需要喜好重定義觸發鍵映射，如 ':map y <Plug>(mapping_name)'。

因此，<Plug> 不過是某個普通按鍵序列的特殊前綴而已，特殊得讓它不可能從鍵盤輸入，主要只用於映射傳遞，同時該中間序列還可取個意義明確好記的名字。一些插件作者爲了進一步避免這個中間序列被衝突的可能性，還在序列中加入插件名，比如改長爲 ":<Plug>(plug_name_mapping_name)'。

不過，映射傳遞鏈可能會引起另一個麻煩。例如請看如下這個映射：

```
: map j gj
: map k gk
```

在打開具有長文本行的文件時，如果開啓了折行顯示選項 (&wrap)，則 gj 或 gk 命令表示按屏幕行移動，這可能比按文件行的 j k 移動更方便。所以這兩個鍵的重映射是有意義的，可惜殘酷的事實是這並沒有達到想要的效果。作了這兩個映射命令之後，若試圖按 j 或 k 時，vim 會報錯，指出循環定義鏈太長了。因爲 vim 試圖作以下解釋：

```
j --> gj --> ggj --> gggj --> ...
```

無盡循環了，當達到一些深度限制後，vim 就不干了。

爲了避免這個問題，vim 提供了另一套命令，在 map 命令之前加上 nore 前綴改爲 noremap 即可，表示不要對該命令的 {rhs} 部分再次解析映射了。

```
: noremap j gj
: noremap k gk
```

當然，前面還提到，良好的映射命令習慣是顯式限定模式，模式前綴還應在 nore 前綴之前，如下表示只在普通模式下作此映射命令：

```
: nnoremap j gj
: nnoremap k gk
```

結論就是：除了有意設計的 <Plug> 映射必須用‘:map’命令外，其他映射儘量習慣用‘:noremap’命令，以避免可能的循環映射的麻煩。例如對本節開始提出的示例規範改寫如下：

```
: nnoremap <Plug>(do_some_thing) :<C-u>call <SID>AFunc()<CR>
: nmap x <Plug>(do_some_thing)
: nmap <C-x> <Plug>(do_some_thing)
: nmap <Leader>x <Plug>(do_some_thing)
```

其中，‘:<C-u>’並不是什麼特殊語法，只不過表示當按下冒號剛進入命令行時先按個 <C-u>，用以先清空當前命令行，確保在執行後面那個命令時不會被其他可能的命令行字符干擾。（比如若不用 nnoremap 而用 noremap 時，在可視模式選了一部分文本後，按冒號就會自己加成‘:‘<,’>’，此時在命令行中先按 <C-u> 就能把前面的地址標記清除。在很小心地用了 nnoremap 時，還會不會有特殊情況導致干擾字符呢，也不好說，反正加上 <C-u> 沒壞處。但若你的函數本就設計爲允許接收行地址參數，則最好額外定義‘:vnoremap’，不用 <C-u> 的版本。）

各種映射命令

前面講了最基礎的‘:map’命令，還有更安全的‘:noremap’命令，以及各種模式前綴限定的命令‘:nnoremap，:inoremap’等。這已經能組合出一大羣映射命令了，不過它們仍只算是一類映射命令，就是定義映射的命令。此外，vim 還提供了其他幾個映射相關的命令。

- 退化的映射定義命令用於列表查詢。不帶參數的‘:map’裸命令會列出當前已重定義的所有映射。帶一個參數的‘:map {lhs}’會列出以 {lhs} 開頭的映射。同樣支持模式前綴縮小查詢範圍，但由於只爲查詢，沒有 nore 中綴的必要。定義映射的命令，至少含 {lhs} 與 {rhs} 兩個參數。
- 刪除指定映射的命令‘:unmap {lhs}’，需要帶一個完全匹配的左參數（不像查詢命令只要求匹配開頭，畢竟刪除命令比較危險）。可以限定模式前綴，如 nunmap {lhs} 只刪除普通模式下的映射 {lhs}。注意，模式前綴始終是在最前面，如果你把 un 也視爲 map 命令的中綴的話。
- 清除所有映射的命令‘:mapclear’。因爲清除所有，所以不需要參數了。當然也可限定模式前綴，如‘:nmapclear’，表示只清除普通模式下的映射。另外還可以有個 <buffer> 參數，表示只清除當前 buffer 內的局部映射。這類特殊參數在下節繼續講解。

特殊映射參數

映射命令支持許多特殊參數，也用 <> 括起來。但它們不同於特殊鍵標記，並不是左參數或右參數序列的一部分。同時必須緊跟映射命令之後，左參數 {lhs} 之前，並用空格分隔參數。

• <buffer> 表示隻影響當前 buffer 的映射，‘:map :unmap’ 與 ‘:mapclear’ 都可接收這個局部參數。

• <nowait> 字面意思是不再等待。較短的局部映射將掩蓋較長的全局映射。

<nowait> 這個參數很少用到。但其中涉及到的一個映射機制有必要了解。假設有如下兩個映射定義:

```
nnoremap x1 something
nnoremap x2 another-thing
```

因爲定義的是兩個按鍵的序列，當用戶按下 x 鍵時，vim 會等待一小段時間，以判斷用戶是否想用 x1 或 x2 快捷鍵，然後觸發相應的映射定義。如果超過一定時間後用戶沒有按任何鍵，就按默認的 x 鍵意義處理了。當然如果後面接着的按鍵不匹配任何映射，也是按實際按鍵解釋其意義。

因此，若還定義單鍵 x 的映射:

```
: nnoremap x simple-thing
```

當用戶想通過按 x 鍵來觸發該映射時，由於 x1 與 x2 的存在，仍然需要等待一小段時間才能確定用戶確實是想用 x 鍵來觸發 simple-thing 這件事。這樣的遲滯效應可不是個好體驗。

於是就提出 <nowait> 參數，與 <buffer> 參數聯用，可避免等待:

```
: nnoremap <buffer> <nowait> x local-thing
```

這樣，在當前 buffer 中按下 x 鍵時就能直接做 local-thing 這件事了。

儘管有這個效用，但 <nowait> 在實踐中還是用得很少。用戶在自行設定快捷鍵時，最好還是遵循 “相同前綴等長快捷鍵” 的原則。也就說當定義 x1 或 x2 快捷鍵後，就最好不要再定義 x 或 x123 這樣的變長快捷鍵了。規劃整齊點，體驗會好很多。當然，如實在想爲某個功能定義更方便的快捷鍵快，可定義爲重複按鍵 xx，因爲重複按鍵的效率會比按不同鍵快一點。(想想 vim 內置的 dd 與 yy 命令)

```
: nnoremap xx most-used-thing
```

另一方面，局部映射參數 <buffer> 卻是非常常用，鼓勵多用。局部映射會覆蓋相同的全局映射，而且當 <nowait> 存在時，會進一步隱藏全局中更長的映射。

• <silent> 在默認情況下，當按下某個映射的 {lhs} 序列鍵中，vim 下面的命令行會顯示 {rhs} 序列鍵。加上這個 <silent> 參數時，就不會回顯了。我的建議是一般沒必要加這個參數禁用這個特性。當映射鍵正常工作時，你不必去理會它的回顯，但是當映射鍵沒按預想的工作時，你就可在回顯中看到它實際映射成什麼 {rhs} 了，這可幫助你判斷是由於映射被覆蓋了還是映射本身哪裏寫錯了。

• <special> 這是相對過時的參數了，它指示當前這個映射命令中接受 <> 標記特殊鍵。在默認不兼容 vi 的設置下，不必加這個參數也能直接用 <> 表示特殊鍵。

• <script> 當堅持用‘:noremap’代替‘:map’這個參數也沒什麼用了。它的本意是限定右參數 {rhs} 不會再與腳本外部的映射相互作用了。

• <unique> 唯一性確保不覆蓋已定義的映射。在使用命令‘:map <unique> {lhs}{rhs}’時，如果發現 {lhs} 在此前已定義，這條重定義映射的命令就會失敗。這個參數一般用在共享插件中，爲了避免覆蓋用戶自己已定義的映射。不過在腳本中，還有兩個函數能作更好的控制。內建函數 mapcheck() 用於判斷一個 {lhs} 是否已被映射，hasmapto() 用於判斷一個 {rhs} 是否有映射過。具體用法請用‘:help’查問相應的函數說明。

• <expr> 這是通過一個表達式間接計算出 {rhs} 的用法。這是個相對高級的用法，將在下一節詳細討論。

* 表達式映射

常規的映射定義‘:map {lhs} {rhs}’只是簡單的將一個鍵序列轉換解析爲另一個序列，所以這是一種靜態的映射。如果在映射定義中結合表達式的思想，通過某種表達式計算出所要轉換的 {rhs}，那就能極大地擴展映射的功能，達到靜態映射所無法實現的靈活性。

有兩種方式在映射定義中使用表達式。一種是 <expr> 參數，另一種是表達式寄存器 @=。我們先討論後一種方式。= 是一種特殊的寄存器，那麼普通的寄存器又是什麼概念呢？那就從宏開始說起吧。雖然乍看之下宏與映射的關係遠着呢，但究其本質也是通過少量按鍵來實現需要大量按鍵的功能。

假設有這麼個需求，將每兩行連接爲一行，怎麼處理比較方便快捷。不妨打開在第一章示例生成的 ~/.vim/vimllearn/helloworld.txt 作爲示例編輯文件吧，如果這個文件你未保存或丟失了，重新生成也是極快的。

vim 普通模式下有個命令 J 用於將光標當前行與下一行連接爲一行，就是刪去其中的回車符。如果光標初始在第一行，那麼 J 就能將第一行與第二行合一行，光標停留在第一行；再按 j 下移到第二行，也就是最初的第三行，再按 J 合併……於是你可用這個按鍵序列 JjJjJjJj... 來將當前 buffer 內的每兩行合併爲一行。

這都是些重複按鍵呀，可以用宏來節省操作呢。假設撤銷剛纔討論的操作，從最初打開的 helloworld.txt 重新開始，（普通模式下）請依次按這些鍵 qaJjq：

• q 是錄製宏的命令，qa 表示將宏保存到寄存器 a 中；

• Jj 就是剛纔我們討論的手動操作，將當前行與下一行合併，再將光標下移一行；

• q 再一個 q 表示結束錄製宏。

現在我們已經有了 a 宏，就可以用 @a 命令播放這個宏了。可見其效果與在錄製時的操作 Jj 是一樣的。然後我們可以進一步在播放宏的命令之前加個重複數字。因爲原來的 helloworld.txt 有 100 行，錄製宏時合了兩行，嘗試播放宏時又合了兩行，所以還需要再合併 48 次。用這個命令 48@a 就可以瞬間將剩余的文本行兩兩合併了。也可以使用 48@@ 命令，因爲 @@ 是表示播放上一次播放過的宏。

（注：上述操作要產生相同結果，需要未打開折行選項，即‘:set nowrap’，或沒有將 j 映射爲 gj 或其他，同時 J 命令也未被映射）

那麼宏到底又是什麼，宏裏面到底保存了什麼神祕的東西。其實它一點都不神祕，宏就是一個寄存器而已。你可以用‘:reg’命令（全名‘:registers’）查看所有寄存器的內容，或者特定地‘:reg a’查看寄存器 a （宏 a）的內容。可見它就是保存着 Jj 這兩個字符而已。可以將它粘貼出來再確認下“o<Esc>“ap’：

• o<Esc> 表示用 o 命令打開新一行，然後用 <Esc> 回到普通模式。如果你按剛纔的批量宏操作後，光標應該位於 buffer 的最後一行；此時在最後新加了一空行，光標也在這空行上。

• “ap 粘貼命令 p 應屬常見，在這之前先按 “a 表示從寄存器 a 中粘貼內容。

執行完這個命令後，就會發現已經將寄存器 a 的內容 Jj 粘貼到當前 buffer 末尾了。常規寄存器有 26 個，即以 a-z 字母命名。我們可以試試其他寄存器，比如先用 v 選定 Jj 這兩個字符，再用命令 “by 將這兩個字符複製進寄存器 b 中。你可以用:reg 命令再次查看下寄存器內容，確認 a 與 b 兩個寄器都保存着 Jj 了。

題外話：我們平時使用複製命令 y 與粘貼命令 p 都不會加寄存器前綴的，這時它們使用的是默認寄存器，其名就是雙引號”，它其實是關聯着最近使用的寄存器，與最近使用那個寄存器內容相同。可以在當前行繼續嘗試 p 命令與 ““p 命令（或在使用每個命令之前先輸入一個空格，分隔內容方便查看），可見它們都粘貼出了 Jj。此外還有大寫字母的寄存器，但它們不是額外的寄存器，只是表示往相應的寄存器中附加內容。比如若 v 選定 Jj 內容後，再按 “Ap ，就表示將這兩字符附加到原來的 a 寄存之後了。可以用‘:reg’查看 a 寄存器的內容已變成 JjJj 了。

爲了說明宏即是寄存器，先用 q! 強制關閉當前的 helloworld.txt 而不保存，再重新打開原始的有 100 行的 helloeworld.txt。如果光標不在首行（vim 有可能會記住光標位置的）則用 gg 回到首行。然後直接用命令 50@b，看看會發生啥。沒錯，這命令也將 buffer 內的文本行兩兩合併了，相當於執行了 50 次 Jj 命令。

所以 @a 或 @b 操作，正式地講不叫“播放”宏，而是“讀取寄存器，將其內容當作普通命令來執行”。其實，當作普通命令來執行的內容，不僅可以放在內部寄存器，也可以放在外部文件中。比如，只將 Jj 這兩個字符保存到一個 Jj.txt 文件中，然後執行 ex 命令‘:source! Jj.txt’。當‘:source’命令之後加個! 符號，就是表示所讀的文件不是當作 ex 命令的腳本了，而是當作普通命令的“宏”了。在這個命令之前，請將光標移到首行，至少不要末行，否則就看不到 j 的效果了。同時由於這個文件只保存了一組 Jj，所以它只合並了兩行。不過普通命令的序列組合可讀性比較差，且很大程度地依賴操作上下文，所以一般不會保存到外部文件，臨時錄製保存到寄存器較爲常見當然你也可以先簡單思考一下如何組織操作序列，明確地寫出來，再複製或剪切到某個寄存器中。

當明白了 @a 的執行意義，也就能更好地理解 @= 的意義了。這裏，= 與 a 一樣是個寄存器，這個特殊寄存叫做表達式寄存器。

請在普通模式下，按下這兩個鍵 @=，此時光標將跳到命令行的位置，不過前面不是:，而是 = 了。vim 在等待你輸入一個有效的表達式，再按回車執行。比如輸入 “Jj”<CR>，這裏 <CR> 表示回車結束輸入並執行，注意 “Jj” 需要引號括起，這樣它纔是個字符串常量表達式，否則若裸用 Jj，回車後 vim 會報錯說 Jj 是個未定義變量。

然後這整個按鍵序列 @="Jj"<CR> 的效果是什麼？就是與普通命令 Jj 一樣，合併兩行並下移。可以用 ':reg' 查看寄存器 = 中的內容也正是 Jj。所以，@= 的意圖是讓用戶臨時輸入一個表達式，vim 將計算該表達式的值，然後將結果值（應是字符串）當作普通命令來執行。如果 @= 之後直接回車，不輸入表達式，則延用原來保存在 = 寄存器中的值。

當你終於明白了 @= 的意義之後，就可以用 @= 來構建表達式映射了（終於回到正題了）。例如：

```
: nnoremap \j @="Jj"<CR>
```

這樣就可以用快捷鍵 j 來"合併兩行並下移"了。當然了，在這個簡單的特定實例中，所謂快捷鍵 j 其實並不比直接輸入 Jj 快多少。那個映射命令似乎也可以直接寫成 ':nnoremap jJj'。然而問題的關鍵是，在 @= 與 <CR> 之間，可以使用幾乎任意合法的 VimL 表達式（即使不是所有），而不會是像 "Jj" 這樣無趣的常量表達式。

舉個實用的例子：

```
:nnoremap <Space> @=(foldlevel(line('.'))>0) ? "za" : "}"<CR>
```

這個映射是說用空格鍵來切換摺疊，即相當於命令 za，但如果當前行根本就沒有摺疊，那就無所謂切換摺疊了，那就換用命令 } 跳到下一個空行。這裏用到了條件表達式?:，我在腳本中很少用這個，不必省 if else 的輸入，但在定義一些映射時條件表達式卻是極簡捷實用的。

在插入模式下（包括命令行模式），不是用 @ 鍵調取寄存器，而是用另一個快捷鍵 <C-R>。比如 <C-R>a 就表示將寄存器 a 的內容插入到當前光標位置上。如果用 <C-R>= 就表示將要讀取表達式寄存器的內容了，此時光標也會跳到命令行處，允許你輸入一個表達式後按回車，vim 就將表達式的計算值插入到光標處。例如：

```
: inoremap <F2> <C-R>=strftime("%Y/%m/%d")<CR>
```

它定義了一個映射，使用快捷鍵 <F2> 在當前光標處插入當前日期（請參閱 strftime() 函數的用法）。

然後再來看 <expr> 參數的意義與用法，比如以下兩個映射定義是等效的：

```
: nnoremap \j @="Jj"<CR>
: nnoremap <expr> \j "Jj"
```

可見，在使用了 <expr> 參數後，@=<CR> 就沒必要了，直接將後面的 {rhs} 參數部分當作一個表達式，vim 首先計算這個表達，然後將其結果值當成真正的 {rhs} 參數來解析爲按鍵序列。

再嘗試將上面那個空格切換摺疊的快捷鍵改寫成 <expr>

```
:nnoremap <expr> <Space> (foldlevel(line('.'))>0) ? "za":"}"
```

（注：我在 vim8.0 中測試該映射有效，但在 vim7.4 中同樣的映射無效，可能在低版本中 <expr> 對條件表達式的?: 的支持不完全，但對於其他簡單表達式無問題）。

除了應用條件表達式，當計算 {rhs} 需要涉及更復雜的邏輯時，還可以包裝在一個函數中，那就幾乎有着無限的可能了。仍以切換摺疊的示例，改寫成函數就如：

```
: function! ToggleFold()
:     if foldlevel(line('.')) > 0
:         return "za"
:     else
:         return "}"
:     endif
: endfunction
:nnoremap <expr> <Space> ToggleFold()
```

不過要注意，VimL 函數的默認返回值是數字 0，如果在函數中忘了返回值，或在某個分支中忘了返回值，那就可能導致奇怪的結果。例如，將上面的 ToggleFold() 函數改寫成：

```
: function! ToggleFold()
:     if foldlevel(line('.')) > 0
:         let l:rhs = "za"
:     else
:         let l:rhs = "}"
:     endif
:     " return l:rhs
: endfunction
:nnoremap <expr> <Space> ToggleFold()
```

假裝忘了返回 l:rhs，那麼快捷鍵 <Space> 將取得 ToggleFold() 的默認返回值 0，就是移到行首的意思了。取消‘:return l:rhs’行的註釋，可使之恢復正常使用。

當然了，用於表達式映射 <expr> 的函數還是有些限制的：

- 不能改變 buffer 內容
- 不能跳到其他窗口或編輯另一個 buffer
- 不能再使用‘:normal’命令
- 雖然可在函數內移動光標，以便實現某些邏輯，但在返回 {rhs} 後會自動恢復光標，所以移動光標是無效的。

總之，映射的表達式函數儘量保持邏輯簡明，以返回一個字符串作爲 {rhs} 爲主，避免在其內執行有其他副作用的操作。更多內容請參考幫助‘:help :map-<expr>’。

* 命令後綴映射

定義命令後綴映射的命令是‘:omap’，當然最好用‘:onoremap’。要能定義有趣的命令後綴映射，首先就要理解命令後綴模式（Operator-pending，直譯操作符懸F模式）。

Vim 普通模式下的許多命令都是“操作符 + 文本對象”範式。比如最常見的 y d c 就是操作符，當你按下這幾個鍵之一後，就進入了所謂的“命令後綴”模式，vim 會等待你輸入後續的操作目標即文本對象。文本對象包括以下兩大類：

1. 使用移動命令後光標掃描過的文本區域，即光標停靠點與原來光標位置之間的區域。

2. 預定義的文本對象，常用的有：

- ap ip 一個段落，段落由空行分隔，ap 包括下一個空行，ip 不包括。
- a(i(或 a) i) 一個小括號，a- 表示包括括號本身，i- 只是括號內部部分。
- a[a] a a，i[i] i i 與小括號類似。
- a” a’ ，i” i’ 與小括號類似，但是由引號括起的部分。

Vim 允許用戶分別獨立定義操作符與文本對象，然後任意組合。命令後綴映射就是可用‘:omap’自定義文本對象。

還是舉個例子。假如你需要經常操作雙引號的字符串，覺得每次用 i” 略麻煩，因爲它實際上是三個鍵，還要按個 Shift 鍵呢。你想選個單鍵來代替這三個鍵，比如說 q 鍵吧。首先，你可能嘗試作如下映射定義：

```
: nnoremap dq di"
: nnoremap cq ci"
```

然而，這只是個普通模式下的映射，並非命令後綴模式下映射，它不具備普適性。這裏只定義了 dq 與 cq 就表明只能用這兩個快捷鍵，但 yq 就無效了（複製字符串?），其他自定義的操作符當然也就無效。

然後試試改成一個命令後綴映射：

```
:onoremap q i"
```

這樣，cq，dq 與 yq 都有效了，如果你知道如何自定義操作符，它對自定義操作符也有效。

一個功能更豐富的例子請參考我寫的一個小插件。在命令後綴模式下，單鍵 q 不僅可以模擬 i” 與 a”，還可以模擬 i(與 a(等括號對象（基於一定的上下文與優先級判斷）。它的映射命令如下：

```
: onoremap q :call qcmotion#func#OpendMove()<CR>
```

不過它所調用的函數實現略複雜，不便全部引用，有興趣的請參閱源代碼。

總結下命令後綴映射的機制，對於‘:onoremap {lhs} {rhs}’映射。首先將 {rhs} 當作普通模式下命令（按鍵序列）執行。如果執行後 vim 仍在普通模式下，且移動了光標，則將前後兩個時刻的光標位置之間的區域當作文本對象。如果執行後在可視模式，則將選擇部分的文本當作文本對象。內置命令 dw dp 類似前一種情況，而 da(，di(類似後一種情況。

命令後綴映射的另一方面是操作符映射。也可以稱之爲命令前綴映射吧。這樣，很多

普通模式下的操作就可理解爲“命令前綴”與“命令後綴”的組合了。定義滿足這樣特性的操作符的映射要分兩步：

1. 設定選項 operatorfunc，其值一般是個函數名，用該函數來執行相應的工作。
2. 用命令 g@ 激活這個函數調用。

當然了，不要將這兩步分開，如果單獨將 operatorfunc 選項設置放在 vimrc ，那就只能定義一個操作符了。最好是類似如下定義：

```
: nnoremap {lhs} :set operatorfunc=OperaFunc<CR>g@
```

就是臨時設定 operatorfunc 的選項值，然後激活它。這樣就能爲不同的 {lhs} 定義不同的操作符了。

操作符函數 OperaFunc() 有一定的規範。它收受的第一個參數表示文本對象的選擇模式（即三種可視模式之一），這個參數是該操作符後面所接的文本對象自動傳遞給它的，其值爲以下三種，在函數內可根據不同值作不同處理：

- “line” 行選擇模式
- “char” 字符選擇模式
- “block” 列塊選擇模式

同時，在該函數內可利用'[與'] 這兩個光標標記（mark）取得所操作文本對象的範圍。即相當於文本對象的選擇範圍，加上參數所指示的選擇模式，就獲得了足夠的信息來操作文本對象了。

縮寫映射

縮寫也是一種映射，不過只用於可輸入模式下。包括插入模式與命令行模式，以及不太常用的替換模式。其命令與映射也類似，不過將 map 換成 abbreviate，如：

```
: abbreviate {lhs} {rhs}
: noreabbrev {lhs} {rhs}
: iabbreviate {lhs} {rhs}
: cabbreviate {lhs} {rhs}
: unabrrev {lhs}
: abclear {lhs}
```

也包括定義（退化參數來列表查詢）、刪除一個、清除所有縮寫的命令。同樣可以用 nore 限定，與模式前綴限制（但只有 i 與 c 分別表示插入模式與命令行模式）。

縮寫的含義是當你輸入 {lhs} 時，自動替換爲 {rhs}。不過由於在插入模式，字符是連續輸入的，所以還有一些限定規則才能讓 vim 識別剛纔輸入的幾個字符是某個縮寫的 {lhs}。

Vim 支持三類縮寫，根據 {lhs} 中關鍵字位置區分。所謂關鍵字就是 iskeyword 選項，一般認爲數字、字符是關鍵字，其他標點符號與空白不是關鍵字。

• 全關鍵字（full-id），即 {lhs} 全部由關鍵字組成。必須完全匹配，即 {lhs} 之前不能有其他關鍵字。

• 關鍵字後綴（end-id），最後一個字符是關鍵字，前面的都不是關鍵字。

• 非關鍵字後綴（non-id），最後一個字符不是關鍵字，前面的可以是任意字符（空格與製表符除外）。

其中，全關鍵字是最常用的縮寫，最直接的想法是用它來糾正拼寫錯誤，如：

```
: abbreviate teh the
: abbreviate higth hight
```

下面兩例是另外兩類縮寫：

```
: abbreviate #i #include
: abbreviate inc# #include
```

在使用縮寫時，還要輸入一個額外的鍵來觸發識別縮寫，這也叫縮寫的展開。一般地，輸入一個非關鍵字後，就會試圖向前回溯尋找是否有縮寫。最常用的是空格與製表符，還有離開插入模式的 <Esc> 與離開命令行模式的 <CR>。當縮寫展開後，這個觸發字符也同時會插入在被展開的 {rhs} 後，如果這不是想用的效果，可用一個快捷鍵 <C-]> 作爲純粹的縮寫展開，而不會插入額外字符。

縮寫同樣支持 <buffer> 與 <expr> 參數。例如：

```
: abbreviate today= <C-R>=strftime("%Y/%m/%d")<CR>
: abbreviate <expr> today= strftime("%Y/%m/%d")
```

這兩個縮寫定義是等效的，在你輸入 “today=’ 之後（再空格或 <C-]> 等觸發）就會替換爲今天的日期。

那麼它與插入模式下的映射又有什麼不同呢：

```
: inoremap <expr> today= strftime("%Y/%m/%d")
```

如果把 “today=’ 定義爲映射的話，那麼在輸入前面幾個字符 “today’ 之前都不會上屏，接着輸入 “=’ 後立即上屏。這個體驗並不好，因爲你即使輸入 “to’ 時，vim 也會等待，根據後續字符才能決定是否當作映射處理。

而定義爲縮寫的話，展開之前的字符是直接上屏的，是否展開的決定延遲，且可由用戶決定是否展開。如果用戶想抑止 “today=’ 的展開，比如確實想在這個字符串之後輸入個空格，則可用 <C-v><Space> 輸入下一個空格。<C-v> 是插入模式下的轉義快捷鍵，它後面接入的按鍵都屏蔽了其特殊意義，就按其字面字符輸入。

結語

使用映射，除了一些基本的命令語法技巧外，更重要的是自己的統一習慣。可以多多凝視一下你的鍵盤佈局，想想定義哪些快捷鍵自己會覺得比較方便與舒服。合適的快捷鍵

對於每個人可能會有不同，不過有些鍵強烈建議不要重映射，請保留其默認意義：

• 數字不要被映射，數字用於表示命令的重複次數。

• 冒號‘:’進入命令行不要改，當然如果覺得冒號不好按，可以將其他鍵也映射爲冒號。兩樣建議保留的鍵是 <Esc> @ 鍵。

• 插入模式下的 <C-v> 與 <C-r>。Vim 的插入模式的默認快捷鍵確實不如普通模式方便，於是有些用戶想把 Emacs 那套快捷鍵映射過來。或者 Window 用戶想將 <C-v> 當作粘貼使用。然後這兩個鍵在 Vim 映射中確實有特殊意義，經常能用來救急，還是保留的好。此外 <C-o> 是臨時回到普通模式使用一個普通命令，也是很有用的，儘可能保留。

另外，關於 <Leader> 的使用。如果基本只用一種映射前綴，使用 <Leader> 是方便的。但如果使用了多個 <Leader> 以對應不同類別的快捷鍵，則不太建議使用 <Leader>，直接寫出映射前綴字符就是。畢竟 mapleader 是個全局變量，若要經常改變其值，就不容易維護了。

除了映射與縮寫，Vim 的自定義命令與自定義菜單的用法與思想也是類似的。自定義菜單是隻用於 gVim 的，本教程不打算介紹，而自定義命令將在一下節介紹。

3.3 自定義命令

命令語法

定義命令與定義映射的用法其實很相似：

```
:command {lhs} {rhs}
```

只不過在使用自定義命令時，{lhs} 是直接輸入到命令行中的，當你按下回車時，vim 就將 {lhs} 替換爲 {rhs} 再執行。所以這在形式上與下面這個映射等效：

```
: nnoremap :{lhs}<CR> :{rhs}<CR>
```

當然，由於‘:command’所支持的參數與‘:map’大相徑庭，並不期望你真的按這方式將自定義命令改成映射。實際上，Vim 的幫助文檔中這樣描述自定義命令的語法的：

```
:command {cmd} {rep}
```

‘:command!’加個歎號修飾則表示重新定義命令 {cmd}，否則若之前已定義 {cmd} 命令，‘:command’原版會報錯。這是爲了保護已定義不被覆蓋，當你確實要覆蓋時，請加! 後綴。在實踐中，一般都是在腳本中定義命令，建議只用! 即可，尤其是在開發階段需要調試腳本時，加上! 方便很多。

大部分命令的! 修飾版都是表示強制執行，忽略錯誤的意思。但上一節介紹的‘:map!’的意義太奇葩，建議直接忘記‘:map!’的用法。

:command 命令的退化用法是一致的：

• :command {cmd} 列出以 {cmd} 開頭的自定義命令；

• :command 列出所有自定義命令；

Vim 的內置命令都是小寫的（除了‘:Next’、‘:X’、‘:Print’），所以要求自定義命令名 {cmd} 只能以大寫字母開頭，其後就類似 VimL 變量名的要求了。然而也不建議在命令名中使用數字，因爲這可能與數字參數混淆。

內置命令可以縮寫（這與上節的縮寫映射不是同個東西），在沒有歧義時，只要輸入命令名的前幾個字母就可以了。自定義命令 {cmd} 同樣可獲得此基本福利。不過內置命令還有更好的福利，就是欽定的縮寫，比如 s 是替換命令 substitute 的縮寫，但它不會與 set 發生歧義，而 set 的縮寫是 se。自定義命令卻無此特性，只能按基本規則，輸入儘可能多的前綴字符來達到唯一確定命令名的目的。不過縮寫只建議在命令行中使用，在腳本中儘量使用全名。

命令屬性

在自定義命令時，可支持多種屬性，就像‘:map’的特殊參數（用 <> 括起來的）。但是在‘:command’中，以一個 - 引導一個屬性（更像 shell 命令行的選項）。所有屬性必須出現在命令名 {cmd} 之前。

• -buffer 局部命令，只能用於當前 buffer。

• -bang 該自定義命令允許有! 後綴修飾。

• -register 第一個參數允許是寄存器名。

• -bar 該自定義命令後面允許用 | 分隔，接續另一個命令。在這種情況下，{rep} 參數內就不能有 | 了，否則會出現解析歧義。

以上這幾個屬性，只有 -buffer 是常用的，並且建議能局部化時儘量局部化。其他的屬性則較少用到。-bang 與 -register 只相當於某種特殊參數，而在同一行中用 | 使用多個語句（命令）的騷操作，能不用盡量不用。

然後，命令還支持幾個複雜的屬性，用 -attribute=value 表示，允許爲屬性指定值，要注意的是等號前後沒有空格，而將整體當作‘:command’命令的一個參數。

• 參數個數，自定義命令 {cmd} 允許多少個參數：◦ -nargs=0 這是默認行爲，不指定該屬性就表示命令不接受參數；

◦ -nargs=1 僅接受一個參數；

◦ -nargs=* 接受 0 或多個參數；

◦ -nargs=? 接受 0 或 1 個參數；

◦ -nargs=+ 接受 1 或多個參數。

按常規用法，多個參數用空格分隔（或製表符）。但如果只有一個參數，末尾的空格會被認爲是參數的一部分。否則若要參數中包含空格，請用 \轉義。

• 範圍數字釋義，是否允許在命令之前加上一個或兩個（以逗號分隔）數字：

◦ -range 允許兩個地址參數或一個數字參數。不加該屬性時，自定義命令默認不接收數字或地址參數。但這只是允許，可選加或不加，也不提供默認數字或地址。

◦ -range=% 允許地址參數，且默認是全 buffer，相當於 1,$。

◦ -range=N 允許一個數字參數，默認是 N，只能用在命令名之前。

◦ -count=N 與 -range=N 類似，不過數字參數不僅可以出現在命令名之前，也可以出現在命令名之後（相當於第一個參數）。-count 與 -count=0 等效。不過注意，-range 屬性與 -count 屬性是互斥的，最好只用其中一個屬性。

• 特殊地址. $ % 所表示的範圍（在允許 -range 時）：

◦ -addr=lines 這也是默認行爲，取當前 buffer 文本行的範圍。

◦ -addr=arguments 指打開 vim 時命令行的文件名參數（其實也可以更改）。

◦ -addr=buffers 指所有打開過的 buffer。

◦ -addr=loaded_buffers 僅指當前加載的 buffer，在某個窗口中顯示的 buffer。

◦ -addr=windows 取所有窗口列表的範圍，僅限當前標籤頁。

◦ -addr=tabs 取所有標籤頁範圍。

注意，-addr 屬性必須要與 -range 聯用纔有意義。它要說明的是當命令的地址參數使用.（當前）$（最後）%（所有）是參照什麼集合而言的。例如定義如下命令：

```
: command -range CmdA {rhs}
: command -range=% -addr=buffers CmdB {rhs}
: command -range=% -addr=tabs CmdT {rhs}
```

則使用命令時，‘:.,$CmdA’ 表示用命令 CmdA 處理當前 buffer 內當前行到最後一行之間的文本行。‘:CmdB’ 表示處理所有 buffer，因爲 -range 的默認範圍是% 表示所有，而 -addr 表示所有的集合是指所有 buffer。同樣，‘:.,$CmdT’ 表示處理從當前標籤頁到最後一個標籤頁，雖然 -range=% 表示默認所有，但使用時可以自己加個特定的地址參數呀。

命令補全

自定義命令還有個最複雜的屬性，是有關補全特性的。值得單獨拿出來討論。

Vimer 初學者傾向於使用映射，可能較少用到自定義命令。但是隨着對 Vim 深入使用與理解，可能就會發覺鍵盤的映射資源是有限的，尤其是要有規律地組織許多容易記住的映射會有瓶頸。這時不妨將眼光投入到自定義命令中。雖然使用命令沒有映射那麼快，但只不過多加冒號與回車，就幾乎有了無限的擴展可能。而且，在命令行中，不僅命令名可以補全，命令參數也可以補全，這就大大減少了記憶負擔。

-complete 屬性就是用於指定命令如何補全參數的，其取值範圍非常廣，這裏僅介紹幾種主要的補全行爲，全部列表請參考‘:help :command-complete’：

• -complete=file 按文件（包含目錄）補全，就像‘:edit’命令按 <Tab> 後會補全文件名那樣。

• -complete=option 補全選項名。

• -complete=help 補全幫助主題。

• -complete=shellcmd 補全外部 shell 可用的命令。

• -complete=tag 補全標籤，類似‘:tag’所需的參數。

• -complete=filetype 補全文件類型名。

總之，如果自定義命令期望它的參數是某一類意義上的參數，就可以指定 -complete 屬性爲相應的值，以方便輸入參數。當然，如果你定義的某個命令要實現比較複雜的功能，vim 預設提供的補全行爲都不滿足要求的話，還可以指定一個函數來實現補全。

- -complete=custom,{func}
- -complete=customlist,{func}

這也叫做自定義補全。要注意的是，“=’與“,’前後都沒有空格，在 custom, 或 customlist, 後直接接一個函數名。

當 -complete 屬性值是 custom 時，函數要求返回一個以回車 \n 分隔的字符串，每一行是一個候選補全項。且 vim 會自動匹配比較光標前已經輸入的部分參數前綴，進行一些過濾。

當 -complte 屬性值是 customlist 時，函數要求返回一個列表，每個元素是候選補全項。但 Vim 不會自動對參數前綴過濾，可能要求用戶自己在函數中過濾。

在這兩種情況，補全函數的定義都是類似的，它應該接收三個參數：

1. a:ArgLead 光標之前的部分參數前綴，
2. a:CmdLine 整個命令行文本，
3. a:CursorPos 當前光標在命令行的位置（按字節計，從 1 開始）。

當用戶按下補全鍵（一般是 <Tab>），Vim 會自動將這三個參數傳給自定義補全函數。用戶在這個函數實現可利用這三個參數所提供的信息（也許不一定要用到全部），返回合適的候選補全項。

命令實現

我們將自定義名之後的 {rep} 參數部分稱爲命令實現。它可以是一串簡單的替換文本，但真正有趣的是它可用一些特殊標記來表示特殊的或動態的內容。這裏的特殊標記也用尖括號 <> 括起，所支持的有意義的標記可能依賴於前面的的命令屬性。

- <line1> <line2> 分別表示地址參數的兩個數字（一般是第一行與最後一行）。含 -range 屬性的命令才能接收這兩個參數。
- <count> 就是由 -count 屬性提供的數字參數。
- <bang> 支持 -bang 屬性的命令，如果使用時加了! 修飾，則在 {rep} 中的
- <bang> 標記轉換爲! 字符，否則就沒任何效果。
- <register> 或簡寫爲 <reg>，支持 -register 屬性的命令，表示可選的寄存器參數；否則也沒任何效果（加上引號“<reg>”才表示空字符串）。
- <lt> 代表左尖括號 <，避免尖括號的特殊意義。比如想在 {rep} 中字面地呈現 <bang> 這幾個字符串，而不是轉化爲! 字符，就可用 <lt>bang>。

先舉個簡單的例子，我們已經知道‘:map!’命令是列出某類映射。雖然上文說過應該忘記這個命令，不過正因爲它安全無害，不妨再拿來作爲演示講解。首先定義這個命令：

```
: command! MAP map
```

這個自定義命令似乎很無趣，不過用大寫版的‘:MAP’代替內置的‘:map’。請試試在命令中輸入‘:MAP’並回車執行，其結果與直接使用‘:map’是一樣的。試試‘:MAP!’呢？Vim 會報錯，說這個命令不支持!。那麼重定義一下這個命令：

```
: command! -bang MAP map<bang>
```

現在，應該‘:MAP’與‘:MAP!’命令都可以使用了，並且分別與‘:map’與‘:map!’等價。這就是 <bang> 用於命令實現參數 {rep} 中的代表意義。同時，如果你沒有定義其他以 MA 開頭的命令，那麼我們這個自定義命令簡寫成‘:MA’或‘:MA!’也是可以的。

由於這個自定義沒有加 -nargs 屬性，默認是不能接收參數的，所以若試圖用‘:MAP lhs rhs’來定義映射會失敗。但是，加了參數屬性後，又如何在 {rep} 中使用相應的參數呢？這就是 <args> 標記的用途，同時這有多個變種：

• <args> 將用戶在自定義命令後輸入的參數原樣替換到 {rep} 中。不過若命令還有 -count 或 -register 屬性的話，前面的屬性應該由 <count> 或 <reg> 捕獲，而 <args> 只表示剩余的參數。

• <q-args> 與 <args> 一樣，先捕獲所有參數，然後將所有參數用引號括起來作爲一個字符串表達式參數。如果沒有參數，這將是一個空字符串（包含引號如 “”）。

• <f-args> 也與 <q-args> 一樣，只不過將捕獲的參數分隔成適用於函數調用時小括號內的參數列表，所以是將每個參數分別引起，並用逗號分隔。這在 {rep} 實現中調用一個函數中非常有用。如果沒有參數，則所調用函數的小括號內也沒有任何東西，即以空參數調用。

現在繼續來改造我們的自定義命令 MAP：

```
: command! -bang -nargs=* MAP map<bang> <args>
```

這樣，‘:MAP’與‘:MAP!’可以繼續用，而且也可以用它來定義映射了，例如：

```
: MAP <buffer> x dd
```

這裏，用自己的‘:MAP’來定義一個映射，將 x 刪除一個字符的功能改爲刪一行。不過由於只爲試驗，所以加 <buffer> 定義成局部映射（注意區別，定義局部命令用-buffer 語法）。

由於我們在定義 MAP 時允許它接收任意個參數 -nargs=*。所以在‘:MAP <buffer>x dd’這個使用場合下，‘:MAP’的所有參數 <buffer> x dd 替換在定義 MAP 時 <args> 的位置上，也就相當於執行‘:map <buffer> x dd’。可以試下執行完，再按 x 是不是實現了預期效果，同時也可以用‘:MAP x’或‘:map x’查看下將 x 定義成啥樣的映射了。

在這個示例中，如果將定義 MAP 時的 <args> 改成 <q-args> 或 <f-args> 的話，結果就不正確了，不能仿擬‘:map’命令了。在實現複雜命令時，後兩個參數變種標記才更有用，作爲函數調用的參數。不過這較爲複雜，留待下一小節再論。這裏先探討一下 <register> 參數的使用，假設繼續爲 MAP 命令添加這個屬性：

```
: command! -bang -register
```

```
            -nargs=* MAP <register>map<bang> <args>
```

先將原來定義的 x 映射刪除：‘:unmap <buffer> x’。然後再用新的‘:MAP’命令定義 x 映射，不過在參數 <buffer> 前額外加個參數 n：

```
: MAP n <buffer> x dd
```

結果是相當於只定義了普通模式下的映射‘:nmap <buffer> x dd’。你可以用‘:map x’查看一下 x 的映射定義確認。並且對比一下‘:MAP <buffer> X dd’不加 n 的用法。

結論就是 <register> 不過是捕獲了第一個參數，<args> 捕獲其他參數。而 MAP 的定義 <register>map<bang> <args> 表明是將第一個參數直接拼在 map 之前作爲映射命令的模式前綴限定，而將其他參數用空格分開後作爲‘:map’命令的參數了。

這樣看來，<register> 似乎很名副其實呀。那麼我們再嘗試下將 un 作爲‘:MAP’的第一個參數，看它會不會變成‘:unmap’用於刪除映射：

```
: MAP un <buffer> x
: MAP un <buffer> X
```

然而，這次 vim 報錯了，提示 umap n <buffer> x 不是一個命令。由些可見，<register> 只捕獲的第一個字母 u，然後將剩余的東西都當成 <args> 了。因爲寄存器名都是一個字母啊。

vim 有些內置命令如‘:del :yank :put’支持後面接一個寄存器名（比如 a），表示對相應的寄存器操作，相當於普通模式的命令 “ad，“ay，“ap。自定義命令就可用 <regsiter> 實現類似的特性，使得自定義命令能像內置命令一樣使用。只不過，<register> 只能捕獲參數中的第一個字母，把它當成是寄存器名，傳給 {rep} 實現部分，卻無法控制 {rep} 如何處理這個字母。因爲‘:map’命令的模式前綴限定恰好也只是一個字母，所以我們的‘:MAP’就可以用 <register> 進行僞裝了。你可以自行嘗試‘:MAP i :MAP c’等用法應該也是有效的。

上一節也提前，使用映射命令，儘量使用更安全的‘:noremap’，所以再重定義命令：

```
: command! -bang -register
           -nargs=* MAP <register>noremap<bang> <args>
```

要測試這個命令是否有效，可定義如下映射：

```
: MAP n <buffer> x xx
```

再按 x 看看是否能正確只刪除兩個字符，還是會發生無盡循環故障（如果有這問題，按 <Ctrl-c> 中斷即可）。

再次提醒：這裏討論不斷優化‘:MAP’命令，只爲說明‘:command’自定義命令的用法與機制。正常使用 vim 下，應該沒必要定義這麼個命令呀。

自定義命令調用函數

除了很簡單的命令，可以調用 vim 既有的內置命令（可能進行必要的包裝修飾）外，大多實用的自定義命令，都是通過調用函數來實現命令要求的功能。這不僅可以實現很複雜的功能，也容易擴展，還使得用法簡明易記，因爲它一般是如下的形式結構之一：

```
:command! {cmd} call WorkFunc(<f-args>)
:command! {cmd} call WorkFunc(<q-args>)
```

當使用自定義命令 {cmd} 時，它後面的命令行參數就會傳入實際工作的函數 WorkFunc() 中。<f-args> 按空格分隔多個參數，然後分別引爲字符串參數傳入，如果要在參數中包含空格，要用 \轉義，要傳入 \就要用兩個反斜槓即 \\。而 <q-args> 則簡單粗暴，將 {cmd} 的所有參數，也就是其後跟着的所有內容當一個字符串參數傳入。在 {cmd} 之後沒有任何參數時，<q-args> 也至少傳入一個空字符串參數（WorkFunc(“”)），但 <f-args> 就不傳入任何參數了（WorkFunc()）。

注意：傳入 WorkFunc() 的參數必定是字符串類型，但由於 VimL 弱類型與自動轉換，如果一個參數像數字，那麼在函數體內將它當作數字處理也完全沒有問題。

按 <f-args> 方式調用函數更爲常見。<q-args> 可能只用於比較特殊的需要，然後要自己在函數體內解析字符串參數。另外，<f-args> 只適用於函數調用參數，用在其他地方的意義不明顯，且易出錯。而 <q-args> 用於函數參數之外也可能是有意義的。本小節暫時不討論 <q-args> 的使用。

使用 range

首先我們需要一個工作函數。不妨複用在 2.4 節講述函數時使用的給文本行編號的示例函數吧，取那個支持 range 特性的版本，並改名爲 NumberLine 重貼於下：

```
1 " File: ~/.vim/vimllearn/fcommand.vim
2 function! NumberLine() abort range
3     for l:line in range(a:firstline, a:lastline)
4         let l:sLine = getline(l:line)
5         let l:sLine = l:line . ' ' . l:sLine
6         call setline(l:line, l:sLine)
7     endfor
8 endfunction
```

然後定義一個命令也叫 NumberLine，用以調用該函數，命令名與函數不需要相同，只是懶得另起名字，同時也想說明，命令與函數重名完全沒問題，因爲它們是完全不是同類概念：

```
: command! -range=% NumberLine <line1>,<line2>call NumberLine()
```

注意到 NumberLine() 函數不支持顯式參數，但可接收隱式的地址參數。而命令‘:NumberLine’正好定義爲支持 -range 屬性，這就要將捕獲的地址參數 <line1>,<line2> 放在 call 之前，由 call 把地址參數傳給 NumberLine() 函數的 a:firstline 與 a:lastline。

現在我們就可以來試用這個自定義命令了。如果直接在命令行輸入‘:NumberLine’回車執行，它會對當前 buffer 的所有文本行編號。因爲 -buffer 屬性的默認值% 就表示所有行，相當於 1,$。如果我們按行可視模式 V 選擇幾行，再按‘:NumberLine’，命令行中實際輸入的是‘:‘<,’>NumberLine’，它就只會對選擇的行進行編號。

使用 count

接着討論下與 -range 相似但互斥的 -count 屬性。<count> 只有一個數字參數，即可放在命令之前，也可以放在命令之後（甚至對是否有空格分隔不敏感）。很多 vim 內置命令的數字表示重複次數，不過在自定義命令中，<count> 只負責捕獲傳遞這個數字參數，並無法控制後續命令如何使用這個數字，就如 <register> 一樣。

我們另外寫個函數，用於對當前行及後面若干行進行相對編號，即當前行號是 0，下一行是 1 等（類似‘:set relativenumber’）。

```
1  function! NumberRelate(count) abort
2      let l:cursor = line('.')
3      let l:eof = line('\$')
4      for l:count in range(0, a:count)
5          let l:line  = l:cursor + l:count
6          if l:line > l:eof
7              break
8          endif
9          let l:sLine = getline(l:line)
10         let l:sLine = l:count . ' ' . l:sLine
11         call setline(l:line, l:sLine)
12     endfor
13 endfunction
14
15 command! -count NumberRelate call NumberRelate(<count>)
```

同時也定義一個相應的命令。試試效果？如果直接運行‘:NumberRelate’，由於 -count 的默認值是 0，所以只對當前行編號爲 0。如果對選區運行‘:‘<,’>NumberRalate’，給命令提供了兩個地址參數？但該命令只接收一個數字參數啊，vim 只會將後面那個地址參數’> 當作數字參數 <count> 傳給函數 NumberRelate() 的參數。同時也可以手動輸入數字如‘:3NumberRelate’或‘:NumberRelate3’都會對當前行及後面 3 行編號。其中 NumberRelate3 的寫法可能會有歧義，如果恰好還有個自定義命名叫叫 NumberRelate3。所以最好用‘:NumberRelate3’來調用。也正是這個原因，不建議在命令名中混入數字。

至於 Vim 爲什麼允許命令與數字參數粘在一起使用，主要是因爲要快捷輸入。很多最常用的命令都是有單字母縮寫的，而與數字參數的組合使用又極頻繁。在這種情況情況下多敲一個空格的性價比太低了（我的命令才一個字母呢），所以就把空格吃了吧。

這個示例也說明，自定義命令調用函數時，參數不一定要用 <f-args> 或 <q-args>，混入其他任何特殊標記也是可以的，只要展開替換後符號函數調用語法即可。再比如，call WorkFunc(<bang>) 是非法的，因爲展開是 call WorkFunc(!)，但 call WorkFunc("<bang>") 是合法的，因爲展開後是 call WorkFunc("!")。而 <count>（其實也包括 <line1> <line2>）可直接放入函數括號內，是因爲它們會展開成一個數字。

使用 f-args

前面兩例所用的函數都不接收參數，如果函數要求參數，就用 <f-args> 傳入吧。假設更改爲文本行編號的需求，在數字編號後還允許加個後綴字符，像 "1."、"1)" 之類的，同時可以定製分隔編號與原文本之間的空格數量。我們重寫 NumberLine 函數，讓它接收兩個參數：

```
function! NumberLine(postfix, count) abort range
    let l:sep = repeat(' ', a:count) "生成字符串：count個空格
    for l:line in range(a:firstline, a:lastline)
        let l:sLine = getline(l:line)
        let l:sLine = l:line . a:postfix . l:sep . l:sLine
        call setline(l:line, l:sLine)
    endfor
endfunction

command! -range=% -nargs=+
      NumberLine <line1>,<line2>call NumberLine(<f-args>)
```

然後也重定義命令 ':NumberLine'，爲其增加 -nargs 屬性，然後用 <f-args> 傳給函數調用。注意雖然可以用 -nargs=1 限定允許一個參數，但不支持 -nargs=2 限定恰好兩個參數，只能用不定數量的 -nargs=* 或 -nargs=+。此時若只用 ':NumberLine' 命令執行，會報錯說參數太少，加上兩個命令行參數後如 ':NumberLine) 4' 就能正常工作了，這表示編號樣式爲 1) 然後接 4 個空格。

注意到 NumberLine() 函數雖然也有個 count 參數。但與上例不同，不能用 -count 屬性與 <count> 參數。首先是因爲 -count 與 -range 屬性只能用一個，不能共存。其次這裏的 count 參數與大多 vim 內置命令對數字參數的解釋很有些不同，只是恰好用了這個形參名而已。因此不要濫用 <count> 參數，能直接用 <f-args> 是最簡潔明瞭的。

如果工作函數 WorkFunc() 沒有 range 屬性，不處理地址範圍的話，那麼自定義命令時，也不要加 -range 屬性，而後面的調用函數寫法也更加簡單。

另外，如果工作函數是腳本作用域的函數，如 s:WorkFunc()，則在 {rep} 部分中調用

寫成 <SID>WorkFunc()，高版本的 vim 也可以直接用 s:WorkFunc()。不過上節的映射命令 ‘:map’，卻只能用 <SID> 而不能用 s:。

* 微命令實例

本節內容所用的命令示例，主要爲闡述概念，也許並無實用性。我在大量使用映射後，也開始對命令有所偏愛了。爲了使命令輸入儘可能方便，我將常用命令也定義很短的幾個大寫字母，並稱之爲 “微命令”。實現腳本放在了 github 上，有興趣的可以參考，傳送門在此。

如果命令名較長，輸入不便時，也可以繼續使用映射來觸發命令，甚至可以將最常用的命令參數也一併包含在映射中。

3.4 execute 與 normal

爲什麼這兩個命令值得單獨拿出來講，因爲它們使得其他大部分 Vim 基本命令變得可編程，用 VimL 編程。不僅是更高層次上的流程控制，更可以控制單個命令的執行，控制所要執行的命令或參數。簡單地說，就是可利用 VimL 語言的一切特性，拼接並生成將要執行的 ex 命令，然後真正執行它。

- :execute 將 VimL 的字符串（值）當作命令執行。
- :normal 用 ex 命令的方式執行普通命令。

基本釋義：execute

還是通過例子來說明。“:execute ‘map x y’” 相當於直接執行命令 ‘:map x y’。當然這似乎沒什麼用，多套層 ‘:execute’ 似乎寫起來還更復雜。但是我們可以這樣寫：

```
: let lhs = 'x'
: let rhs = 'y'
: execute 'map ' . lhs . ' ' . rhs
```

似乎還更復雜了是不？然而，這背後的思想在於，lhs 與 rhs 都是變量，我們可以根據需求計算出它們值，然後再定義相應的映射。這就可以靈活地動態地執行 ex 命令了。一般情況下，我們會把 ‘:execute’ 命令寫在腳本或函數中，比如寫個叫 s:BuildMap() 的函數封裝一下：

```
1 function s:BuildMap() abort
2     let l:lhs = 'x'
3     let l:rhs = 'y'
4     let l:map = 'nnoremap'
5     execute l:map . ' ' . l:lhs . ' ' . l:rhs
6     " 或者下面這行語句等效
```

```
7       execute l:map l:lhs l:rhs
8   endfunction
```

‘:execute {expr}’ 這是 ‘:execute’ 的正式語法，它後面接一個表達式。vim 首先計算出這個表達式的值，一般期望它是個字符串，如果不是字符串也會自動轉爲字符串。然後執行這個字符串。

事實上它可以跟多個表達式，‘:execute {expr1} {expr2}’，vim 會先求出各個表達式的值，再拼接成一個字符串，中間有個空格。如果你不確定這個自動拼接機制，或者不想在相鄰表達式之間多加個空格，則可以用 VimL 的字符串連接操作符，一個點號 “.’，這樣就可以自己把握要或不要這個空格了。

在上個示例中，我們將函數內的局部變量直接賦值了（常量字符串），這僅爲說明 ‘:execute’ 的用法特徵。更好的封裝做法是利用函數參數，例如：

```
1   function s:BuildMap(map, lhs, rhs) abort
2       execute a:map a:lhs a:rhs
3   endfunction
```

把函數體簡化爲一條語句了。當然更健壯的做法應該先檢測一下 a:map 參數是否爲合法的映射命令，以避免一些災難錯誤。而且真正的映射命令可能還不止由這三部分組成，還可能有很多類似 <buffer> 這樣的特殊參數呢，不過這裏暫不考慮了。

當把眼光向外拓展，函數參數怎麼來? 那就把 VimL 當作普通腳本語言（類似 python，perl，lua 這種腳本思想），根據需求計算變量的值，傳遞參數調用函數就可以了。

基本釋義：normal

那麼 ‘:normal’ 命令又有何妙用。因爲 VimL 本質上只是 ex 命令的組合，原則上在 vim 腳本中只能使用 ex 命令。但是 Vim 的基本模式是普通模式，有很多基本操作在普通模式用普通命令可以很方便地達成，但在 ex 命令行模式（或腳本中）卻可能一時找不到對應的命令來實現相同功能，或者可以實現卻寫起來麻煩。

這時 ‘:normal {commands}’ 命令就來幫忙了。它將其後的 {commands} 參數當成是在普通模式下按下的字符（鍵）序列來解釋。比如我們知道在普通模式下用 gg 跳到首行，用 G 跳到末行。可有什麼 ex 命令來完成這任務嗎? 有肯定有，至少可以調用函數 cursor() 來放置光標，但是用 ‘:normal’ 似乎更簡明：

```
: normal gg
: normal G
```

要注意與 ‘:execute’ 命令不同的是，‘:normal’ 的參數 {command} 它不是個表達式，它就表示字面上看到的字符。如果寫成:normal “gg” 反而錯了，因爲在普通模式下，前兩個字符（按鍵）“g 是取寄存器 g 的意思呢。

‘:normal! {commands}’ 的歎號變種，表示後面的 {commands} 不受映射的影響。因爲正常用戶使用 vim 時都會在 vimrc 中定義相當多的映射，所以 ‘:normal’ 命令會繼續根據映射來再次查尋將要執行的（普通）命令。這往往使得結果不可預測，所以一般情況下建議使用 ‘:normal!’ 而非 ‘:normal’。

不過，使用 ‘:normal’ 還是有些限制的，畢竟不能完全像普通模式那樣的使用效果。最重要的一點是 ‘:normal’ 命令必須完整。如果命令不完整，vim 自動在最後添加 <Esc> 或 <Ctrl-c> 返回普通模式，以保持完整性。完整性不太好定義，那就舉例說幾個不完整的：

• 操作符在等待文本對象時不完整。如果執行 ‘:normal! d’ 什麼事都不會發生。因爲在普通模式下 d 會等待用戶繼續輸入文本對象。而用 ‘:normal’ 來執行時，就無從等待，結果就是像按下 d 後又按下 <Esc> 取消了。但是 ‘:normal! dd’ 能正確完成刪除一行的操作。

• 用 ‘:normal’ 命令進入插入模式操作後，會自動 <Esc> 回到普通模式，不會停留在插入模式。例如 ‘:normal! Ainsert something’ 會在當前行末增加一些字符串，但是整個命令結束後，不能期望它還在插入模式，它會回到普通模式。

• 在 ‘:normal’ 後面用冒號進入命令行模式並輸入一些命令，卻不能以想當然的方式執行。比如輸入 ‘:normal! :map’ 後按回車，它並不會執行 ‘:map’ 命令列出映射。因爲它相當於在命令行輸入 ‘:map’ 後按 <Ctrl-c> 取消了，並不是按回車執行了。你必須用個技巧將回車符添加到 ‘:map’ 之後才行，直接按回車是執行 ‘:normal!’ 這條命令的意思。這樣輸入：‘:normal! :map^M’ 再按回車就可以了，其中 ^M 表示回車符，通過按 <C-v><CR> 兩個鍵才能輸入。

總之，‘:normal’ 命令執行完畢後，會保證仍回到普通模式。也因此不能通過 Q 鍵進入 Ex 模式。

execute + normal 聯用

正如上面看到，‘:normal’ 命令後的參數（普通命令按鍵序列），只適於可打印字符，對於特殊字符，須用 <C-v> 轉義後才能輸入，這不太方便。但是可用 ‘:execute’ 命令再套一層，因爲它接收的字符串表達式，當用雙引號引起字符串時，特殊字符可用 \轉義。比如爲解決上面那個難題 ‘:normal! :map’：

```
: execute 'normal! ' . ":map\<CR>"
```

但是，execute + normal 的基友組合，遠不止是爲了輸入特殊字符這麼簡單。‘:execute’ 還可以使 ‘:normal’ 也用上變量。例如，我們可以用 5gg 來跳到第 5 行，用 ‘:normal’ 命令也能跳到特定行：

```
: normal! 5gg
: normal! 10gg
```

然而，你無法直接動態地改變 5 或 10 這個數字，借且 ‘:execute’ 就可以了：

```
: let count = 15
: execute 'normal! ' . count . 'gg'
```

再舉個例子，在第 1.2 節，我們在普通模式下生成了一個滿屏盡是“Hello world!”的文章，回顧如下：

```
20aHello World!<ESC>
yy
99p
```

現在，我們用 VimL 語言編程的思路，利用 execute + normal 重新生成。既是編程，封裝成函數纔好：

```
1 function HelloWorld(row, col) abort
2     normal G
3     let l:word = 'Hello World!'
4     for i in range(a:row)
5         normal! o
6         execute 'normal! ' . a:col . 'a' . l:word
7     endfor
8 endfunction
```

函數接收兩個參數，分別表示生成多少行，與每行多少個“Hello World!”。在函數體中，‘:normal! G’先將光標定位到當前 buffer 末尾，以便在末尾插入許多“Hello World!”。然後對每一行循環，每行循環中，先用 o 命令打開新行，再用‘:execute’拼接重複多次的 a 命令。

你可以用函數調用命令‘:call HelloWorld(100,20)’來達到 1.2 節的效果，並且可調用行列數生成不同規模的“Hello World!”。

* 用 execute 定義命令

在上一節中，我們推薦了一種定義命令的常用範式：call WorkFunc(<f-args>)。這裏再介紹另一種定義命令的有趣範式：

```
:command! {cmd} execute ParseFunc(<q-args>)
```

形式上只是把‘:call’命令換成了‘:execute’命令。將自定義命令 {cmd} 的所有參數打包傳給函數 ParseFunc()，期望它返回一個字符串，再用‘:execute’執行它。

這另有什麼妙用呢？一般情況下，用‘:execute’可能只想到用它來執行常規的 ex 命令，但是也並不妨礙它用於執行 VimL 的特殊語法命令。例如，‘:let’命令只能一次創建一個變量，下面這種“連等號”的語法是錯誤的：

```
: let x = y = z = 1
```

但我們可以試着自定義一個‘:LET’命令，讓它允許這個語法：

```
" File: ~/.vim/vimllearn/clet.vim
function! ParseLet(args)
    let l:lsMatch = split(a:args, '\s*=\s*')
    if len(l:lsMatch) < 2
        return ''
    endif
    let l:value = remove(l:lsMatch, -1)
    let l:lsCmd = []
    for l:var in l:lsMatch
        let l:cmd = 'let ' . l:var . ' = ' . l:value
        call add(l:lsCmd, l:cmd)
    endfor
    return join(l:lsCmd, ' | ')
endfunction

command! -nargs=+ LET execute ParseLet(<q-args>)
```

這代碼有點長，適合保存在腳本文件中再‘:source’。先解釋下函數 ParseLet() 的意思：它首先將輸入參數按等號（兩邊允許空格）分隔成幾部分；將最後部分當作是值，其余每部分當作一個變量，然後構造命令用‘:let’爲每個變量賦相同的值；最後將幾個賦值語句用 | 連接並返回，| 是在同一行分隔多個語句的意思。

有了 ParseLet 函數後，再定義一個命令‘:LET’，現在就可以嘗試下連續賦值了：

```
: LET x = y = z = 1
: echo x
: echo y z
: echo ParseLet('x = y = z = 1')
```

可見 x y z 三個變量都已經被賦值爲 1 了。最後一個‘:echo’語句是爲了顯示‘:LET’如何工作的，實質上它轉化爲 let x=1 | let y=1 | let z=1 多個賦值語句了。

那麼，新定義的‘:LET’能否正確處理變量的作用域呢，我們寫個函數測試一下：

```
function! TestLet()
    LET l:x = y = z = 'abc'
    echo 'l:x =' l:x 'x =' x
    echo 'l:y =' l:y 'y =' y
    echo 'l:z =' l:z 'z =' z
endfunction

```

```
8 call TestLet()
9 echo 'x =' x 'y =' y 'z =' z
```

我們在函數中也定義了 x y z 這三個局部變量。結果表明，用‘:LET’定義的局部變量與全局變量也互不衝突的，可放心使用。

不過，‘:execute’命令畢竟還是有所限制的。只適合用於定義一些簡單的“宏命令”，並不能妄圖重定義一些複雜的語法結構。而且，‘:execute’的效率也不高。

3.5 * 自動命令與事件

前面章節介紹了自定義快捷鍵（:map）與自定義命令（:command），這都是響應玩家的主動輸入而快速做些有用的工作。這也算是對 Vim 的 UI 設計吧。誰說只有圖形界面纔算 UI 呢，況且在 gVim 中的自定義菜單，也確實與自定義命令或映射很相似呀。

本節要介紹的自動命令，卻是讓 Vim 在某些事件發生時自動做些工作，而不必再手動激活命令了。當然了，自動命令在生效前，也是需要定義的。

自動命令的定義語法

自動命令用‘:autocmd’這個內置命令定義，它至少要求三個參數：

```
: autocmd {event} {pat} {cmd}
```

- {event} 就是 Vim 預設的可以監測到的事件，比如讀寫文件，切換窗口等。
- {pat} 這是模式條件的意思，一般指是否匹配當前文件。
- {cmd} 就是事件發生且滿足條件時，要自動執行的命令。

在一個命令中可以有多個事件，事件名用逗號分開，且逗號前後不能有空格。模式也可能以逗號分隔爲多個模式。因爲 {event } 與 {pat} 都相當於是‘:autocmd’的單個參數，其內不能有空格。但最後部分 {cmd} 可以有空格。

一般情況下，{cmd} 就是合法的 ex 命令，將它拷貝到命令行也能手動執行那種。不過 {cmd} 中可能含有一些特殊標記 <> ，在執行前會替換成實際值，這才大大增加了自動命令的靈活性，而非只能執行靜態命令。

在 vim 內部，相當於爲每個事件 {event} 維護了一個列表，每當用‘:autocmd’爲該事件定義了一個自動命令，就將這個命令加到列表中。然後每當事件發生，就遍歷這個命令列表，如果它滿足相應的 {pat} 條件，就會執行這個 {cmd} 命令。

因此，每發生一個事件，vim 都可能自動執行許多命令。就比如文件類型檢測與語法高亮着色，就是通過自動命令實現的。當你安裝一些複雜插件，可能會自動執行更多的命令。而我們自己用‘:autocmd’定義的自動命令，只是添加在原來的命令列表之後，做些自定義的額外工作。

與此前的‘:map’與‘:command’一樣，退化的‘:autocmd’是查詢功能：

- :autocmd {event} {pat} 列出與事件及模式相關的自動命令。

- :autocmd * {pat} 列出滿足某個模式的所有事件的自動命令。
- :autocmd {event} 列出與某事件相關的所有自動命令，不論模式。
- :autocmd {event} * 與:autocmd {event} 等效，* 就表示匹配所有。
- :autocmd 列出所有自動命令。

歎號修飾的‘:autocmd!’命令用於刪除自動命令，參數意義與退化命令一樣：

- :autocmd! {event} {pat} 根據事件與模式刪除自動命令。
- :autocmd! * {pat} 只根據模式條件刪除自動命令。
- :autocmd! {event} 只根據事件刪除命令。
- :autocmd! {event} * 只根據事件刪除命令。
- :autocmd! 刪除所有自動命令。

但是，歎號也可以修飾完整的非退化的‘:autocmd’，就如定義自定義模式一樣：

```
: autocmd! {event} {pat} {cmd}
```

它表示先將滿足事件 {event} 與模式 {pat} 的所有自動命令刪除，然後添加自動命令 {cmd} 。因此這是覆蓋式的定義自動命令，此後，在滿足相應事件與模式時，就只會執行這一個自動命令了。依前文介紹，在定義命令與函數時建議用覆蓋式的歎號修飾命令‘:command!’與‘:function!’。但對於自動命令，還是慎重用覆蓋式的‘:autocmd!’，因爲可能無法從本條語句判斷會覆蓋掉什麼自動命令。

自動命令組

自動命令組 augroup 是組織管理自動命令的有效手段。爲理解自動命令組是有必要的，先回顧上一小節所介紹的自動命令機制，在未利用命令組的情況下，會發生什麼不良後果。

因爲‘:autocmd’定義自動命令時是將其添加到自動命令列表末尾的，所以如果在腳本如 vimrc 中定義了自動命令，隨後又重新加載了該腳本，那自動命令列表中就會出現兩項重複的自動命令了。對於某些“安全”的自動命令，重複執行不外是浪費效率而已，但有些自動命令在第二次執行卻有可能引發錯誤呢。

其次，用‘:autocmd!’刪除自動命令時，它是刪除所有自動命令。即使加了事件與模式兩個限制條件，也無法避免影響擴大化，因爲別的插件或 Vim 官方插件也可能爲相同的事件與模式定義的一些有用的自動命令啊。

爲了解決這個管理問題，引入了自動命令組的概念。自動命令組名字是用以標記一個自動命令組的符號，取名規則就按 VimL 變量名的規範吧（雖然幫助文檔中說似乎可以用任意字符串作爲組名，除了空白字符），不要用奇怪的字符，同時也是大小寫敏感的。然後兩個特殊的自動命令組名 END 與 end 是保留的，有着特殊意義。

在不發生理解歧義下，我們就用自動命令組名錶示一個自動命令組吧，且在本節中，不妨用“組名”來作爲自動命令組名的簡寫吧。

於是，在定義自動命令的‘:autocmd’命令中，還支持一個可選的組名參數，它緊接命令之後，而在 {envent} 事件之前：

```
: autocmd [group] {event} {pat} {cmd}
```

正因爲組名是‘:autocmd’的第一個參數，可有可無，當省略時，第一個參數就是事件名了。所以我們選取組名時，還要避免與事件名（這是 Vim 預設的範圍集）重名，以避免歧義。

在定義自動命令時，如果指定了 [group] 組名參數，就表示將所定義的自動命令添加到這個自動命令組中。你可以認爲每個組都爲不同事件維護了不同的自動命令列表，同一事件在不同組內關聯着各自不同的命令列表。

對於刪除自動命令的‘:autocmd!’變異命令，也同樣支持在第一個參數中插入可選的組名。在指定組名後，就表示只刪除該組內的自動命令（當然可再限定事件與模式）。

那麼，在缺省組名參數時，‘:autocmd’與‘:autocmd!’又怎樣工作的呢。其實它是針對當前組添加或刪除自動命令的。那麼當前組又是什麼東西呢? 它是用‘:augroup’命令選定的:

```
: augroup {name}
```

在執行這個命令之後，{name} 就是當前組名了。當 {name} 組名此前尚不存在時，也會自動創建一個組，然後再選擇這個組作爲當前組。此後‘:autocmd’或‘:autocmd!’若不指定組名參數，就用 {name} 替代了。

那麼，在第一次使用‘:augroup’選定當前組名之前，當前組又是什麼呢? 那就是默認組（default group）了。默認組沒有名字，你要把它想象爲空字符串也行。或者形式地說，默認組名是 END 或 end，因爲在以下命令表示選擇默認組名:

```
: augroup END
```

因此，在腳本中定義自動命令的一般規範是這樣的:

```
1 augroup SPECIFIC_GROUP
2     autocmd!
3     autocmd {event} {pat} {cmd}
4 augroup END
```

首先選定一個組，緊接着用‘:autocmd!’刪除該組內原來所有舊的自動命令，然後用‘:autocmd’重新定義新的自動命令，可能有多條‘:autocmd’自動命令，最後用 END 選回默認的（無名）組。這樣，即使這個腳本重新加載，這個組內的自動命令也正是在這塊腳本內所能看到的這些自動命令了。

當然了，你的組名不要別的組衝突。建議依據腳本文件名或插件名定義組名，且用大寫字母，因爲組名很重要，但其實又不必寫很多次，故用大寫字母表示合適。而且，儘量把自定義命令寫在一塊，不要分散。

這樣，在組內定義的自動命令就有了局部特性，相當於局部自動命令，而在組外的（無名默認組）自動命令，就相當於全局自動命令。在編程的任何時刻，都儘量用局部的東西，

少用全局的東西。就自動命令而言，除了直接在命令行臨時測試下什麼自動命令，在腳本插件中，永遠不在默認的無名“全局”組定義自動命令。

另外提一點，退化的查詢命令‘:autocmd’在缺省組名參數時，不是依據當前組，而是列出所有組內的自動命令。這與定義或刪除自動命令時的缺省行爲不同。這也好理解，因爲只是查詢，還是希望儘可能查出更多，而修改操作，卻要儘可能縮小影響範圍。

還有，組名隻影響定義與刪除自動命令的操作，但不影響事件觸發自動命令。即不管定義在哪個組內，事件觸發時，並且檢測滿足模式後，就能執行相應的自動命令。

使用事件

Vim 會監測大量事件，詳細列表請查看文檔‘:help autocmd-events’，這裏只介紹幾種常用的事件。事件名不分大小寫，然而建議按文檔中的名字使用事件。

• 讀事件。有很多相似但略有細微差別的事件，BufNewFile 指創建新文件，BufRead 指讀入文件。一般用這兩個就可以了。若有更多控制需求，可用 BufReadPre 與 BufReadPost，這些事件一般會在‘:edit’等命令時觸發。若用‘:read’命令，可觸發 FileReadPre 與 FileReadPost 事件。

• 寫事件。:w 寫入當前文件時觸發 BufWrite 事件，部分寫入（如’<,’>w file ）則觸發 FileWrite 事件。

• 窗口事件。新建窗口觸發 WinNew，進入窗口觸發 WinEnter，離開窗口前觸發 WinLeave 事件。

• 標籤頁事件。類似窗口事件有 TabNew TabEnter TabLeave。

• 整個編輯器啓動與離開事件：VimEnter VimLeave。

• 文件類型事件，當 &filetype 選項被設置時觸發 FileType。

舉些例子。爲了方便，直接在命令行中定義自動事件了，只爲簡單測試。不過首先也創建一個組吧，比如：

```
: augroup TEST
: augroup END
```

在這裏，先是創建並選定 TEST 爲當前組，然後什麼也沒干又用 END 選回默認組。此後我們定義自動命令時都將顯式地指這在 TEST 組上操作。你也可以先不用‘:augroup END’，保持當前組爲 TEST，只爲了想在之後的‘:autocmd’缺省組名？但是在命令行操作中說不定會觸發加載其他插件，這樣就會改變當前組名了。所以爲了原子操作的獨立性，還是先選回默認組吧，也避免後來忘了執行‘:augroup END’。

然後定義一個自動命令：

```
: autocmd TEST BufNewFile,BufRead * echomsg 'hello world!'
```

這裏顯式指定在 TEST 組內定義自動命令，‘:autocmd’只能使用已存在的組，所以我們之前纔要用‘:augroup TEST’然後又‘:augroup END’的“空操作”。BufNewFile

與 BufRead 經常同時用，這樣不管是打開編輯已存在的文件，還是新建文件都能觸發。在 {pat} 部分我們先簡單用 * 表示匹配所有。最後的 {cmd} 部分僅是打印一條消息。

現在請試試打開另一個文件，或切換另一個 buffer，看看會不會打出 “Hello World!” 的消息。如果消息被其他後續消息覆蓋而看不到，請用 ‘:message’ 打開消息區（可能還須用 G 翻到最後）再看是否有這個記錄。

再定義另一個自動命令，在打開 vim 腳本文件中顯示不同的消息：

```
: autocmd TEST BufNewFile,BufRead *.vim echomsg 'hello vim!'
```

然後用 ‘:e $MYVIMRC’ 打開你的啓動配置文件，看看有什麼歡迎消息？似乎仍是打印 “Hello World!”，而不是 “Hello vim!”？那麼請用 ‘:echo $MYVIMRC’ 查看下你的配置文件是哪個文件，一般應該是 ~/.vimrc 或 ~/.vim/vimrc，它並不是以.vim 作爲後綴的文件名呢。所以不能匹配 *.vim 這個模式。

那麼手動打開一個確實以.vim 爲後綴的文件再試試看吧，或者新建一個 vim 文件 ‘:e none.vim’。不出意外的話，你應該會看到兩條消息，“Hello World!” 與 “Hello vim!” 都打印了，因爲它確實同時滿足剛纔定義的兩個自動命令啊，所以兩個都執行了。然後再試試 ‘:e none.VIM’，新建一個文件以大寫的 “.VIM’ 爲後綴名。這也不會觸發 “Hello vim!”，可見文件模式是區別大小寫的，它未能匹配到 “.VIM’。關於模式的細節，下一小節再詳敘。

爲了避免消息太多，我們先把剛纔兩個自動命令刪除了，再定義另外一個自動命令：

```
: autocmd! TEST
: autocmd TEST BufNewFile, BufRead * echomsg
          'hello ' . expand('<afile>')
```

這裏，<afile> 表示在觸發自動命令時，所匹配的那個文件名（一般是當前文件名）。再試試打開文件，會打印什麼歡迎消息？

切記：在用 autocmd! 刪除命令時，要加上組名 TEST，否則可能會刪去一些定義在默認組的自動命令。

寫文件事件也一樣定義自動命令：

```
: autocmd TEST BufWrite * echomsg 'bye ' . expand('<afile>')
```

然後隨便編輯一個文件，用 ‘:w’ 寫入，是否能預期的 “bye ...” 消息。很可能看不到的。因爲 BufWrite 事件是在開始寫的時刻觸發，然後寫完後 vim 一般會自動再打印另一條消息顯示寫入多少字節。消息被覆蓋了！但用 ‘:message’ 再翻到末尾應該就能看到了。那麼我們把事件改爲寫之後試試：

```
: autocmd TEST BufWritePost * echomsg
          'goodbye ' . expand('<afile>')
```

再看看寫文件時會提示什麼消息。順便說一下，BufWritePre 事件與 BufWrite 其實是等效的。如果沒有特殊需要，建議用 BufWrite 比較簡便。

然後再舉個切換窗口的自動事件:

```
: autocmd TEST WinEnter * echomsg 'Enter Window: ' . winnr()
: autocmd TEST WinLeave * echomsg 'Leave Window: ' . winnr()
```

這裏 winnr() 函數將取得當前窗口編號。定義完這兩個自動事件後，請將你的 vim 分裂出多個窗口，在窗口間切換，以及關閉多余窗口，看看會有什麼消息提示（用‘:message G’確認消息)。由此你應該能得到結論，切換窗口時先觸發 WinLeave 事件，再觸發 WinEnter 事件。

其他事件就不一一舉例了，請自行對感興趣的事件進行測試。然後在實際寫插件或腳本時，若想實現某個自動功能，先查閱文檔，找個合適的事件，理解它的觸發時機。如果 Vim 沒有提供合適的事件，可能自動命令就無能爲力了。不過幸運的是，Vim 已經提供了大量的事件，應該能滿足絕大部分需求了。或者，當你功夫足夠深時，可以從近似的事件入手進而曲線救國。

再次提醒，如果是在腳本中定義自動命令，請按以下規範寫:

```
1  " save in somefile.vim
2  augroup TEST
3      autocmd!
4      autocmd BufNewFile,BufRead * echomsg
5              'hello ' . expand('<afile>')
6      autocmd BufWrite * echomsg
7              'bye ' . expand('<afile>')
8      autocmd BufWritePost * echomsg
9              'goodbye ' . expand('<afile>')
10 augroup END
```

在‘:augroup’塊內不必再指定 TEST 組名了，雖然也可以在每個‘:autocmd’命令重複加上這個組名，但是建議省略。因爲萬一以後因爲某種原因要改組名，卻忘記了同步修改裏面的每個組名，那就麻煩了。

所以，把‘:augroup’與‘:augroup END’當作像‘:function!’與‘:endfunction’一樣的獨立單元塊吧。只不過裏面的命令不是由顯式的‘:call’調用，而是 vim 根據事件自動調用了。於是，很顯然地，自動命令組名應像（全局）函數名一樣，不要與其他組名衝突。

在實用的自動命令中，{cmd} 部分一般是調用一個工作函數，以簡化‘:autocmd’的語法，而把複雜的邏輯實現放在函數中。特殊標記如 <afile> 表示匹配的文件名，在觸發自動命令時才展開。但有個例外，<sfile> 表示的是定義該自動命令時所在腳本文件（假設你不是把自動命令放在函數中定義，一般應該是這樣)。同時，在 {cmd} 部分也可以用 <SID> 表示當前定義腳本範圍的元素，比如 s:Function。

文件模式

定義自動命令時 ‘:autocmd’ 的第二參數 (可選組名除外)，即 {pat} 是文件模式的意思。它不同於正則表達式，而像是操作系統的文件名通配符。即 * 表示任意字符，? 表示單個字符。詳細符號意義請查看 ‘:help file-pattern’。這裏只強調幾點需要注意的地方：

• 逗號表示多個模式的或意義。如 *.c,*.h,*cpp 表示 c/c++ 文件。

• 如果模式中沒有路徑分隔符 /，則只匹配文件名。

• 如果模式中包含 / 則要匹配文件全路徑名。如 /vim/src/*.c 只匹配位於 /vim/src/ 目錄下的 c 文件，這可能是 Vim 源代碼的工程文件。而 */src/*.c 則匹配任意目錄下的子目錄 src/ 內的 c 文件，可能表示任意一 c 語言工程內的源文件。

• 一些命令如 ‘:edit’ 會將其參數內的環境變量 (如 $MYVIMRC) 與特殊寄存器 (如% 與 #) 展開，則在將實際文件名展開後再匹配自動命令中的文件模式。

如果文件模式 {pat} 用一個特殊參數 <buffer> 代替，則表示定義了一個只局部於特定 buffer 的自動命令。這又有幾個變種：

• <buffer> 所定義的自動命令影響當前 buffer，即只有在當前 buffer 才能觸發。

• <buffer=N> 這裏 N 是一個數字，表示隻影響編號爲 N 的 buffer。用 ‘:ls’ 命令或 bufnr() 函數可以查看 buffer 的編號，那算是唯一不變的 id。

• <buffer=abuf> 這裏的 <abuf> 是在觸發自動命令時的特殊標記，如同 <afile> 表示觸發的文件，而 <abuf> 表示觸發的 buffer 編號。這個參數只在當自動命令中定義另一個自動命令時有用。

例如，‘:autocmd BufNewFile * autocmd CursorHold <buffer=abuf> echo ‘hold’’ 表示每當新建一個文件 (BufNewFile 事件) 時，就爲該文件 buffer 定義一個自動命令，該自動命令的意圖是每當 CursorHold 事件觸發 (光標停留一段時間)，就打印一個消息。

相對之下，<buffer> 參數更簡單易懂，如該參數能滿足局部自動命令的要求，優先使用這個吧。例如，將 ‘:autocmd {event} <buffer>’ 命令放在某個函數內，先通過其他命令切換到正確的 buffer 內，再調用這個函數爲該 buffer 定義局部自動命令。由於這已經是局部自動命令了，加不加組名的影響都不那麼大了。

其他提示

• 自動命令是相對高級的功能，可用 has(’autocmd’) 判斷你的 Vim 版本是否已編譯了這個功能，或 ‘:version’ 看輸出是否有 +autocmd。

• 文件類型檢測的自動命令定義在 filetypedetect 組內，當你想創造新文件類型時，也可往這個組內添加自動命令，如 ‘:autocmd filetypedetect *.xyx setfiletype xfile’。但沒事不要誤用 ‘:autocmd!’ 刪除這個組內的其他自動命令。

• 嵌套的自動命令。默認情況下，自動命令中使用的命令如 ‘:e’、‘:w’ 不再繼續觸發讀寫事件，但是加上 nested 可選參數，可允許嵌套。如 ‘:autocmd {event} {pat} nested {cmd}’ 使得在執行 {cmd} 時有可能繼續觸發自動命令 (不過有最大嵌套層數限制，除非必要，慎用)。nested 可選參數應位於 {cmd} 之前，只有保持 {cmd} 在最後部分，才方便

在自動命令使用必要的空格啊。

• 自動命令也可以手動調用，當你覺得有這需求時再去查文檔吧，‘:doautocmd’與‘:doautoall’。

• 太多自動命令有可能降低效率，因此有個選項 &eventignore 可以指定忽略某些事件。這不會刪除自動命令，但有些事件不會觸發了，相應自動命令也就不會執行了。在一個命令之前附加‘:noautocmd {cmd}’可臨時使得本次執行 {cmd} 時不會觸發自動命令。如‘:noautocmd w’在這次寫入過程中，不會觸發寫事件。

3.6 * 調試命令

對任何一門語言，都有必要掌握調試技巧或手段。本節介紹 VimL 語言編程可以怎麼調試，介紹一些自己的經驗與體會。

echo 大法

對於不太龐大的程序或腳本，在關鍵疑點處打印消息都是簡單方便的發現問題的手段，姑且也算一種調試方法吧。

不過這明顯有個問題，當程序調試完畢後，這些只爲調試用的 echo 打印命令留着很礙事呀，可能會與正常的輸出混雜在一起，干擾正常結果呢。所以最好是能將正常的 echo 與調試的臨時 echo 區分開來。正好，VimL 有個奇葩規定，在每行行語句之前的: 冒號是可選的。這是爲了與命令行表觀上一致，然而正常的 vim 腳本一般都不會自找麻煩多加這個冒號。但是若按語法規則，你在每行語句之前加一個冒號（甚至多個冒號）都是沒有關係的。

於是，不妨自己規範一下，將調試用的打印語句，都寫成‘:echo’，或者喜歡多個空格‘: echo’也行。而在正常的程序輸出語句中，則用整潔的無冒號 echo 版。這樣，當調試完畢，確認程序無誤後，就可以用 vim 強大的編輯命令將這些調試命令都刪了:

```
: g/:\s*echo/delete
```

當然，你也許並不是想徹底刪除，只是想註釋掉，那就可用替換命令:

```
: g/:\s*echo/s/:\s*echo/" echo/
```

當‘:s’命令使用的正則表達式與前面的‘:g’命令的正則表達式是一樣的時候，可以簡寫成‘:g/:\s*echo/s//“ echo/’。因爲‘:s//replace/’命令中，空模式的意圖是重複使用上次的模式（寄存器 / 的內容）。若是爲達這個目的，直接用替換命令也可以的‘: %s/:\s*echo/“ echo/’。不過與‘:g’命令聯用（先查找目標行，再替換）會更靈活點，比如想將首列替換爲註釋符”，而不影響內縮進的‘:echo’命令，則可使用這樣的替換命令:

```
: g/:\s*echo/s/^./"/
```

如果想更細緻點，可以自行將‘:echo’與‘::echo’用於不同場合，比如不同等級的調試輸出。

還有個問題，‘:echo’命令的輸出是易逝的，後一批的命令（vim 的解釋單元）輸出會覆蓋掉前一批的命令輸出。如果想保存這樣的輸入，有以下幾種辦法：

• :echomsg 用這個命令替換‘:echo’，則輸出信息會保存在消息區，以後可用‘:message’再次查看，當消息區的信息比較多時，可能需要翻頁查看，G 跳到最後一頁，基本上就是最近的輸出了。

• :redir 命令重定向，可以將隨後的‘:echo’消息重定向至文件、寄存器、變量中，當然也會同時顯示在屏幕上。不再需要重定向功能時用‘:redir END’命令取消。

○ - :redir! > {file} 重定向到文件中，當文件已存在時，用! 強制覆蓋。

○ - :redir @{reg}> 重定向至寄存器，如果支持系統剪貼板，用 * 或 + 表示。

○ - :redir => {var} 重定義向至一個變量中。

○ - :redir » 將上述命令中的 > 換爲 » 表示附加。

• &verbosfile 將詳情信息寫入這個選項值指定的文件中。&verbose 選項值設定詳情信息的等級。

斷點進入調試模式

Vim 也提供了正式的調試模式，那有點像允許單步執行的 Ex 模式。一般需要先設置斷點，隨後當腳本運行到斷點處，就進入了調試模式。添加斷點用‘:breakadd’命令：

• :breakadd file [lnum] {name} 在一個 vim 腳本文件中的某行加斷點，行號可選。注意如果提供行號，行號參數位於文件名之前，如果省略行號，相當於第 1 行。隨後當‘:source {name}’加載該腳本時，執行到那行時會暫停，進入調試模式。

• :breakadd func [lnum] {name} 在某函數的第幾行打斷點。{name} 指函數名。如果是全局函數，那就是直接的函數名，如 FuncName。如果是腳本局部函數，如 s:FuncName，則要先找到那個腳本在當前 vim 會話的腳本號（:scriptnames），然後實際的函數名是 <SNR>dd_FuncName，其中 dd 就是腳本號數字。如果是匿名函數，它沒有名字，就只能用其函數編號了，如‘:breakadd func 1 21’表示在第 21 個匿名函數的第 1 行處打斷點。那匿名函數編號如何確定呢？如果這個函數有出錯了，在錯誤信息中會打印出出錯函數的名字與行號，匿名函數沒名字就用編號代替了。（沒有出錯麼？沒出錯爲啥調試？）至於 [lnum] 行號，可理解爲函數體內相對於函數頭定義的相對行號，可不是該函數定義塊在腳本文件中的行號。即從函數定義頭按 [lnum] 次 j 就是函數斷點處。

• :breakadd here 當你在編輯一個 vim 腳本文件時，相當於在當前文件的當前行加入斷點。如果你已經進入了調試模式，並且已經單步進入了某個函數，‘:breakadd here’也可以在當前函數的當前行加入斷點，下次再次調用該函數時（或下次循環）運行到此處時也會暫停。

當用‘:breakadd’添加了一些斷點後，可用‘:breaklist’查看斷點信息。也可用‘:breakdel’刪除斷點。

• :breakdel {nr} 按斷點號刪除某個斷點（:breaklist 會列出斷點號）。

• :breakdel * 刪除所有斷點。

• :breakdel file [lnum] {name}

• :breakdel func [lnum] {name}

• :breakdel here 這三個命令與‘:breakadd’相似，但是刪除斷點。

除了通過‘:breakadd’添加斷點，以期將來運行到彼處時進入調試模式外，還有另外兩種方式直接進入調試模式：

• :debug {cmd} 在執行命令之前附加‘:debug’，就將在執行該命令時立即進入調試模式，一般接着用 s（step in）深入調試，如果用 n（step over）可能就將整條 {cmd} 命令當作一步直接執行完了，並不能達到調試效果。

• vim -D {other args} 在啓動 vim 時，通過 -D 命令行參數，直接在加載 vimrc 時就開始進入調試模式了。

調試模式

調試模式是一種特殊的 Ex 模式，除了一般的 ex 命令，還可以使用以下調試命令：

• cont (c)，表示繼續執行，直到遇到下一斷點，或結束。

• quit (q)，中斷，類似 <Ctrl-C>

• interrupt (i), 也類似 <Ctrl-C>

• next (n)，單步執行，類似 step over，會跳過函數調用與加載文件。

• step (s)，單獨執行，類似 step in，會步進函數調用或加載文件。

• finish (f)，結束當前加載腳本或函數調用，回到調用處。

• backtrace (bt) 或 where，顯示調用堆棧。

• frame (fr) {N}，切換到堆棧的第 N 層，可用 +，- 表示相對層。

• up / donw, 在堆棧處上移一層（fr +1）或下移一層（fr -1）。

以上這些調試命令可以儘可能縮寫，只要前綴字符不衝突（小括號裏也已標出最簡縮寫）。直接敲回車表示重複上一次命令，這樣就不必每次輸入 s 或 n 命令了。

調試命令沒有補全功能，只有普通 ex 命令才能補全。如果要使用與調試命令相同的普通 ex 命令，多加一個冒號，如‘:next’。但是，由於在 Ex 模式，編輯窗口是不更新的（事實上，只要調試過程稍長，vim 窗口就完全被調試信息覆蓋了），很多普通 ex 命令是沒有效果後，只有在完成調試模式後重回普通模式才能反映編輯窗口的變化。

真正有價值的 ex 命令是可用 echo 命令查看變量值，並且能根據當前環境查看相應作用域的變量值，比如在加載腳本時可查看 s:var，運行到函數內部可看局部變量 l:var（在函數內默認局部變量，‘:echo var’就相當於‘:echo l:var’）。而在正常的命令行下面，是無法查看 s:var 與 l:var 變量的。

在調試模式中，只能打印出正要執行的那行的源代碼。這是典型的命令行式的調試方式，並不能像 IDE 那般分裂出源碼窗口，直接將光標定位到正在執行的行上。如果想查看完整代碼，只能用另外一個 vim 打開源文件查看了（有可能出現 *.swp 衝突問題，用只讀模式打開就好）。所以 VimL 調試的可視化程序仍稍嫌不足，希望日後還有改進。

第四章　VimL 數據結構進階

在第 2.1 章已經介紹了 VimL 的變量與類型的基本概念。本章將對變量類型所指代的數據結構作進一步的討論。

4.1　再談列表與字符串

引用與實體

前文講到，列表作爲一種集合變量，與標量變量（數字或字符串）有着本質的區別。其中首要理解的就是一個列表變量只是某個列表實體的引用。

直接用示例說話吧，先看數字變量與字符串變量的平凡例子：

```
: let x = 1
: let y = x
: echo 'x:' x 'y:' y
: let y = 2
: echo 'x:' x 'y:' y
:
: let a = 'aa'
: let b = a
: echo 'a:' a 'b:' b
: let b = 'bb'
: echo 'a:' a 'b:' b
```

我們先創建了一個數字變量 x，併爲其賦值爲 1，然後再創建一個變量 y，併爲 x 的值賦給它。顯然，現在 x 與 y 的值都爲 1。隨後我們改變 y 的值，重賦爲 2，再查看兩個變量的值，發現只有變量 y 的值改變了，x 的值是沒改變的。因此，即使在創建 y 變量時用‘:let y = x’看似將它與 x 關聯了，但這兩個變量終究是兩個獨立不同的變量，唯一有關聯的也不外是 y 初始化時獲取了 x 的值。此後這兩個變量分道揚鑣，可分別獨立地改變運作。對於字符串變量 a 與 b，也是這個過程。

然後再看看列表變量：

```
: let aList = ['a', 'aa', 'aaa']
```

```
: let bList = aList
: echo 'aList:' aList 'bList:' bList
: let bList = ['b', 'bb', 'bbb']
: echo 'aList:' aList 'bList:' bList
```

結果似乎與上面的數字或字符中標題很相似，沒什麼差別嘛。雖然 bList 一開始與 aList 表示同一個變量，但後來給 bList 重新定義了一個列表，也沒有改變原來的 aList 列表。這與字符串 a，b 的關係很一致呢。

但是，我們重新看下面這個例子：

```
: unlet! aList bList
: let aList = ['a', 'aa', 'aaa']
: let bList = aList
: echo 'aList:' aList 'bList:' bList
: let bList[0] = 'b'
: echo 'aList:' aList 'bList:' bList
```

這裏先把原來的 aList bList 變量刪除了，以免上例的影響。仍然創建了列量變量 aList，與 bList 並讓它們 “相等”。然後我們通過 bList 變量將列表的第一項 [0] 改成另一個值 b，再查看兩個列表的值。這時發現 aList 列表也改變了，與 bList 作出了同樣的改變，兩者仍是 “相等”。

通過這組試驗，想說明的是，當 VimL 創建一個列表（變量）時，它其實是在內部維護了一個列表實體，然後這個變量只是這個列表實體的引用。命令 “:let aList = [‘a’, ‘aa’, ‘aaa’]” 相當於分以下兩步執行工作：

1. new 列表實體 = [‘a’ , ‘aa’ , ‘aaa’]
2. let aList = 列表實體的引用

然後命令 ‘:let bList = aList’，它只是將 aList 變量對其列表實體的引用再賦值給變量 bList，結果就是，這兩個變量都引用了同一個列表實體，或說指向了同一個列表實體。而命令 ‘:let bList[0] = ‘b’’ 則表示通過變量 bList 修改了它所引用的列表的第一個元素。但變量 aList 也引用這個列表實體，所以再次查看 aList 時，發現它的第一個元素也變成 ‘b’ 了。實際上，不管是對 aList 還是 bList 進行索引操作，都是對同一個它們所引用的那個列表實體進行操作，那是無差別的。

對於普通標量變量，則是另一種情況。當執行命令 ‘:let b = a’ 時，變量 b 就已經與 a 是無關的兩個獨立變量，它只是將 a 的值取出來並賦給 b 而已。但 ‘:let bList = aList’ 是將它們指向同一個列表實體，在用戶使用層面上，可以認爲它們是同一個東西。但是當執行 ‘:let bList = [‘b’, ‘bb’, ‘bbb’]’ 後，變量 bList 就指向另一個列表實體了，它與 aList 就再無聯繫了。

可見，當對列表變量 bList 進行整體賦值時，就改變了該變量所代表的意義。這時與對字符串變量 b 整體賦值是一樣的意義。然而，標量始終只能當作一個完整獨立的值使用，

它再無內部結構。例如，無法使用 let b[0] = 'c' 來改變字符串的第一個字符，只能將另一個字符串整體賦給 b 而達到改變 b 的目的。

總結，只要牢記以下兩條准則：

- 標量變量保存的是值；
- 列表變量保存的是引用。

函數參數與引用

我們再通過函數調用參數來進一步說明列表的引用特性。舉個簡單的例子，交換兩個值，可以引入一個臨時變量，由三條語句完成：

```
: let tmp = a
: let a = b
: let b = tmp
```

這種交換值的需求挺常見的，考慮包裝成一個函數如何？

```
1 function! Swap(iValue, jValue) abort
2     let l:tmp = a:iValue
3     let a:iValue = a:jValue
4     let a:jValue = l:tmp
5 endfunction
```

但是，當嘗試調用 ':call Swap(a, b)' 時，vim 報錯了。因爲參數作用域 a: 是隻讀變量，所以不能給 a:iValue 或 a:jValue 賦另外的值。但是，即使參數不是隻讀的，這樣的交換函數也是沒效果的（比如用 C 或 python 改寫這個交換函數）。因爲在調用 Swap(a, b) 時，相當於先執行以下兩個賦值語句給參數賦值：

```
: let a:iValue = a
: let a:jValue = b
```

此外，不管在函數內不管怎麼倒騰參數 a:iValue 與 b:jValue，都不會影響原來的 a 與 b 變量。因爲如前所述，標量賦值，只是拷貝了值，等號兩邊的變量是再無聯繫的。

但是，交換列表不同位置上的元素是可實現的，比如把上面那個交換函數改成三參數版，第一個參數是列表，跟着兩個索引：

```
1 function! Swap(list, idx, jdx) abort
2     let l:tmp = a:list[a:idx]
3     let a:list[a:idx] = a:list[a:jdx]
4     let a:list[a:jdx] = l:tmp
5 endfunction
```

請試運行以下語句確認這個函數的有效性：

```
: echo aList
: call Swap(aList, 0, 1)
: echo aList
```

在寫較複雜的 VimL 函數時，一般不建議在函數體內大量使用 a: 作用域參數。因爲傳入的參數是無類型的，很可能是不安全的。最好在函數的開始作一些檢查，合法後再將 a: 參數賦給一個 l: 變量，然後在函數主體中只對該局部變量操作。此後，如果入參的需求有變動，就只修改函數前面幾行就可以了。例如再將交換函數改成如下版本：

```
1  function! Swap(list, idx, jdx) abort
2      if type(a:list) == v:t_list || type(a:list) == v:t_dict
3          let list = a:list
4      else
5          return " 只允許第一參數爲列表或字典
6      endif
7
8      let i = a:idx + 0 " 顯式轉爲數字
9      let j = a:jdx + 0
10
11     let l:tmp = list[i]
12     let list[i] = list[j]
13     let list[j] = l:tmp
14 endfunction
```

再用以下語句來測試修改版的交換函數：

```
: call Swap(aList, 1, 2)
: echo aList
```

可見，即使在函數體內，將參數 a:list 賦給另一個局部變量 l:list，交換工作也正常運行。因爲 g:aList a:list 與 l:list 其實都是同一個列表實體的引用啊。

列表解包

在 3.4 節我們用 execute 定義了一個‘:LET’命令，用於實現連等號賦值。但實際上可以直接用列表賦值的辦法實現類似的效果。例如：

```
: LET x=y=z=1
: let [x, y, z] = [1, 1, 1]
: let [x, y, z] = [1, 2, 3]
```

其中前兩個語句的結果完全一樣，都是爲 x、y、z 三個變量賦值爲 1。注意等號左邊也需要用中括號把待賦值變量括起來，分別用等號右側的列表元素賦值。這種行爲就叫做列表解包（List unpack），即相當於把列表元素提取出來放在獨立的變量中。顯然用這種方法爲多個變量賦值更具靈活性，可以爲不同變量賦不同的值。

這個語法除了可多重賦值外，還能方便地實現變量交換，如：

```
: let [x, y] = [y, x]
```

用過 python 的對此用途應該很有親切感。不過在 VimL 中，等號兩邊的中括號不可省略，且等號兩邊的列表元素個數必需相同，否則會出錯。不過在左值列表中可以用分號分隔最後一個變量，用於接收右值列表的剩余元素，如：

```
let [v1, v2; rest] = list
相當於
let v1 = list[0]
let v2 = list[1]
let rest = list[2:]
```

在上例中假設 list 列表元素只包含簡單標量，則解包賦值後，v1、v2 都是隻接收了一個元素值的標量，而 rest 則接收了剩余元素，它還是個（稍短的）列表變量。而 list[2:] 的語法是列表切片（slice）。

索引與切片

這裏再歸納一下列表的索引用法：

- 索引從 0 開始，不是從 1 開始。
- 可以使用負索引，-1 表示最後一個索引。
- 可以使用多個索引，這也叫切片，表示列表的一部分。

要索引一個列表元素時，用正索引或負索引等效的，這取決於應用場合用哪個方便。如果列表長度是 n，則以下表示法等效：

```
list[n-1] == list[-1]
list[0]   == list[-n]
list[i]   == list[i-n]
```

然而，不管正索引，還是負索引，都不能超出列表索引（長度）範圍。

列表切片（slice）是指用兩個索引提取一段子列表。list[i:j] 表示從索引 i 到索引 j 之間（包含兩端）的元素組成的子列表。注意以下幾點：

- i j 同樣支持負索引，不管用正負索引，如果 i 索引在 j 索引之後，則切片結果是空列表。
- 如果 i 超出了列表左端（0 或 -n），或 j 超出列表右端，結果也是空列表。

• 可省略起始索引 i，則默認起始索引爲 0；省略結束索引 j，則默認是最後一個索引 -1；如果都省略，只剩一個冒號，list[:] 與原列表 list 是一樣的（但是另一個拷貝列表）。

• 可以爲切片賦值，即將一個列表的切片放在等號左邊作爲左值，可改變索引範圍內的元素值，但一般右值要求是與切片具有相同項數的列表。

• 不支持三索引表示步長，list[i:j:step] 或 list[i:j:step] 在 VimL 中是非法的，不支持跳格切片，只支持連續切片。

• list[s:e] 表示法有歧義，因爲可能存在腳本局部變量 s:e，則用該變量值單索引列表。可在冒號前後加空格避免歧義，list[s : e] 表示切片。

處理列表的內置函數

VimL 提供了一些基本的內置函數用於列表的常用操作，詳細用法請參考文檔:help list-functions，這裏僅歸納概要。

• 查詢列表信息的函數：
  ◦ len(list) 取列表長度，列表的最大索引是 len(list)-1。
  ◦ empty(list) 判斷列表是否爲空，即列表長度爲 0。
  ◦ get(list, i) 相當於 list[i]，但是當 i 超出索引範圍時，get() 函數不會出錯，且
  ◦ 可再提供第三參數表示超出索引時的默認值（如果省略，默認值 0）。
  ◦ index(list, item) 查找一個元素在列表中的位置，如果不存在該元素，則返回 -1。
  ◦ count(list, item) 檢查一個元素在列表中出現多少次。
  ◦ max(list) min(list) 查詢一個列表中的最大或最小元素。
  ◦ string(list) 將列表轉化爲字符串表示法。
  ◦ join(list, sep) 將列表中的元素用指定分隔符連接爲一個字符串表示。
• 修改列表元素的函數：
  ◦ add(list, item) 在列表末尾添加一個元素。
  ◦ insert(list, item) 在列表頭部添加一個元素，比 add() 尾添加低效。但 insert() 可額外提供第三參數表示要插入的索引位置，省略即 0 表示插在最前面。
  ◦ remove(list, idx) 刪除位置 idx 上的一個元素，remove(list, i, j) 刪除從 i 到 j 索引之間的所有無素，相當於 unlet list[i:j]。
• 生成列表的函數：
  ◦ range() 支持一至三個參數，生成連續或定步長的數字列表。
  ◦ extend(list1, list2) 連接兩個列表，相當於 list1+list2，但 extend 會原位修改 list1 列表。與 add() 函數不同的是，add 只增加一個元素，而 extend 是加入另一個列表。
  ◦ repeat(list, count) 相當於不斷連接自身，總計重複 count 次，生成一個更長的列表。
  ◦ copy(list)，生成一個列表副本，用等號賦值只是引用同一個列表實體，用 copy() 函數才能生成另一個新列表（每個元素值與原列表相同而已）。copy() 函數是淺拷貝，列表元素直接賦值。如果要考慮列表元素也可能是列表或字典（引用），則用 deepcopy(list) 遞歸拷貝完全的副本。
  ◦ reverse(list) 將一個列表倒序排列，原位修改原列表。

- split(list, pattern)，將一個字符串分解爲列表，相當於 join() 的反函數。

• 分析列表的高階函數：

- sort(list) 爲一個列表排序。
- uniq(list) 刪除列表中相鄰的重複元素，列表需已排序。
- map(list, expr) 將列表每個元素進行某種運算，將結果替換原元素。
- filter(list, expr) 將列表每個元素進行某種運算，若結果爲 0，則刪除相應元素。

這些高階函數，除了都會原位修改作爲第一個參數的列表外，都還能接收額外參數表明如何處理每個元素。由於額外參數可以是另一個函數（引用），所以稱之爲高階函數。其具體用法略複雜，在後面相關章節將繼續講解部分示例。

字符串與列表的關係

字符串在很大程序上可以理解爲字符列表，可以用類似的索引與切片機制。但是，字符串與列表的最大區別在於，字符串是一個完整的不可變標量。所以，凡是可以改變列表內部某個元素的操作（如索引賦值、切片賦值）或函數（如 add/remove 等），都不可作用於字符串。而 copy() 也沒必要用於字符串，直接用等號賦值即可。不過 repeat() 函數作用於字符串很有用，能方便生成長字符串。

將字符串打散爲字符數組，可用如下函數方法：

```
: let string = 'abcdefg'
: let list = split(string, '\zs')
: echo list
```

split(string, pattern) 函數是將字符串按某種模式分隔成列表的。\zs 不過是一種特殊模式，它可以匹配任意字符之間（詳情請參考正則表達式文檔），所以結果就是將每個字符分隔到列表中了。

4.2 通用的字典結構

爲什麼說字典是通用結構。因爲在 VimL 中字典是最複雜的內置類型了，而更復雜的數據結構都能以字典爲基礎構建出來。不過從基礎的概念上理解，字典與列表其實也有些相似之處，當掌握了列表之後，對字典的用法也就容易了。

字典與列表的異同

在其他一些（腳本）語言中，對應 VimL 的列表與字典的概念，也叫數組與關聯數組。所以字典也可以看成是一種特殊的列表，無序的以字符串爲索引的列表。字典的索引也叫鍵，在 VimL 中，字典的鍵只能是字符串，當數字用作字典鍵時也被隱式轉爲字符串。其它類型的值，一般不能用作字典的鍵。

請看這個示例：

```
let list = range(10)
let dict = {}
for i in range(10)
    let dict[i] = i
endfor

echo 'list =' list
echo 'dict =' dict

for [k, v] in items(dict)
   echo k v
endfor
```

用 range() 創建了一個列表 list，包含的元素是 0-9 這十個數字。然後創建了一個空字典 dict，再用循環爲字典增加鍵值，也用相同的 0-9 這十個數字作爲鍵與值。這樣，在表觀上，dict 與 list 似乎保存着相同的元素，用相同的索引能得到相同的值，比如 dict[5] 與 list[5] 都得到數值 5。但通過:echo dict 可以發現，dict 字典的鍵，其實不是數字，而是字符串 ('0', '1' 等)。

爲理解遍歷字典的範式 for [k, v] in items(dict)，可先用 echo items(dict) 查看這是什麼。可見 items(dict) 返回一個列表，該列表的每個元素又是個小列表，包含鍵與值兩個元素。所以你明白了，字典的元素，不像列表的元素那麼簡單的一個值，而是一個 "鍵值" 對。所謂關聯數組名稱也源於此，每個鍵對應一個值。鍵是唯一的，但值可不唯一，即不同鍵可關聯相同的值。

在字典循環中，也用到了上節介紹的列表解包的多重賦值的功能，相當於如下語句:

```
: let [k, v] = items(dict)[0]
: let [k, v] = items(dict)[1]
: ...
: let [k, v] = items(dict)[9]
```

也因此，[k, v] 必須用中括號括起來。

在這個特殊的例子中，遍歷字典所得的值也許是與列表一樣有序。但請記住，字典不保證有序。同時，在一般應用中，最好不要用連續的數字作爲字典的鍵，那應該直接使用列表更高效且方便。但如果是很稀疏的有大量空洞的列表，則用字典或許是有意義的。如:

```
: let dict[10] = 'a'
: let dict[100] = 'b'
: let dict[1000] = 'c'
```

這樣，只爲 dict 增加了三個元素。但若爲 list[1000]='c' 賦值，則會爲列表增加 1000 個元素，中間的無用索引都浪費了。

在常用使用字典時，建議用簡單字符串索引，所謂簡單字符串，即是可充當 VimL 標記符（變量名）的字符串。這時，字典的中括號索引可用點索引簡化寫法，即 dict['name'] 可簡化等效於 dict.name。當索引是一個字符串變量時，用中括號索引更方便，即 dict[varname]。

還有一點需要重點理解的是，字典變量與列表變量一樣，是引用而已。請看以下示例：

```
: let d1 = {}
: let d2 = {}
: echo d1 == d2
: echo d1 is d2
: let d3 = d1
: echo d3 is d1
```

雖然 d1 與 d2 都是空字典，它們按值比較是一樣的，但其實是不同的字典實體，用 is 比較顯示不一樣。爲另一個變量 d3 賦值後，就指向相同的字典實體了。

操作字典的內置函數

上節介紹的許多關於列表的函數，也可作用於字典。只是其他參數意義可能不一樣，對於字典時，一般是根據鍵來處理的。詳情請查閱 ':h dict-functions'。

但以下幾個函數是字典特有的：

- has_key(dict, key) 檢查一個字典是否含有某個鍵。
- keys(dict) 返回由字典的所有鍵組成的列表。
- values(dict) 返回由字典的所有值組成的列表。
- items(dict) 返回由字典的所有鍵值對組成的列表。

這幾個返回列表的函數一般用於 for ... in 循環中。字典的內部存儲是無序的，但可用 for ... in sort(keys(dict)) 根據鍵順序遍字典。

4.3 嵌套組合與擴展

VimL 雖然只提供了列表與字典兩種數據結構，但通過列表與字典的合理組合，幾乎能表達任意複雜的數據結構。這與許多其他流行的腳本語言（如 python）的思想如出一轍。本節就討論在 VimL 中如何用列表與字典表示常用數據結構。

棧與隊列

棧是所謂後進先出的結構，隊列是先進先出的結構。這可以直接用一個 list 表示，因爲 list 相當於個動態數組，支持隨意在兩端增刪元素。

如果只在列表尾部增刪元素，那就實現了棧行爲。如果尾部增加而在頭部刪除，就實現了隊列行爲，如：

```
function Push(stack, item)
    call add(stack, item)
endfunction

function Pop(stack)
    call remove(stack, -1)
endfunction

function Shift(queue)
    call remove(stack, 0)
endfunction
```

在這個示例中，用 Push/Pop 表示棧操作，用 Push/Shift 表示隊列操作。這隻爲簡明地說明算法意圖，實際應用中最好先檢查 stack/queue 是否爲 list 類型，以及檢查列表是否爲空。

鏈表

在腳本語言中，其實根本不用實現鏈表，因爲動態數組本身就可用於需要鏈表的場合。在 VimL 中，就直接用 list 表示線性鏈就夠了。除非你真的需要很頻繁地在一個很長的 list 中部增刪元素，那麼或可用字典來模擬鏈表的實現。

例如，以下代碼構建了一個有 10 個結點的鏈表，每個結點是個字典，value 鍵表示存儲內容，next 表示指向下一個結點：

```
let head = {}
for value in range(10)
  let node = {'value': value, 'next': head}
  let head = node
endfor
```

其實在上面的循環中，臨時變量 node 可以省略。head 始終指向鏈表的起始結點，可通過 next 鍵依次訪問剩余結點，末尾結點的 next 鍵值是空字典。

這裏的關鍵是，字典的值，或列表元素的值，不僅可以存儲像數字與字串符的簡單標量，還可以存儲另一個列表或字典（的引用）。基於這樣的嵌套與組合，就可以表達更復雜的數據結構了。

二維數組（矩陣）

如果列表的每個元素都是另一個列表，那就構成了一個二維數組。例如：

```
let matrix = []
for _ in range(10)
  let row = range(10)
  call add(matrix, row)
endfor
```

構建了一個 10x10 大小的矩陣，其中每個行向量由 range(10) 生成。這樣快速生成的矩陣每一行都相同，或許不是很有趣，但是可以用以下兩層循環重新賦值：

```
for i in range(10)
  for j in range(10)
    let matrix[i][j] = i * j
  endfor
endfor
```

從數學意義上的矩陣講，它應是規整的矩形，即每行的長度是一樣的。但當在 VimL 中用列表的列表表示時，其實並不能保證每一行都等長。例如：

```
let text = getline(1, '$')
for i in range(len(text))
  let line = text[i]
  let text[i] = split(line, '\s\+')
endfor
```

在這裏，首先用 getline() 獲取當前 buffer 的所有行，保存在 text 這個列表變量中，其中每個元素表示一行文本字符串。在隨後的循環中，又將每行文本分隔成一個個單詞（空格分隔的字符串），將標量字符串元素轉化爲了另一個列表。因此，text 最終結果就是列表的列表，即二維數組。而一般情況下，每行的單詞數量是不等，所以這個二維數組不是規整的矩陣。

事實上，這個示例的循環可以直接用 map() 函數代替：

```
: let text = getline(1, '$')
: call map(text, "split(v:val, '\\s\\+')")
```

樹

以二叉樹爲例，也可用一個字典來表示樹中某結點，除了需要一個鍵（如 value）來保存業務數據，還用一個 left 鍵表示左孩子結點，right 表示右孩子結點，這兩個應該都是另一個具有相同結構的字典引用，如果缺失某個孩子，則可用空字典表示。

```
: let node = {}
```

```
: let node.value = 0
: let node.left = {}
: let node.right = {}
```

這樣，只要有一個字典變量引用了這樣的一個結點（不妨稱之爲根結點），就相當於引用這一棵樹，沿着結點的 left 與 right 鍵就能訪問整棵樹的所有結點。兩個子結點都是空字典時，該結點就是所謂的葉結點。

不過，由於每個結點含有兩個方向的子結點，要遍歷樹可不是那麼直觀。有興趣的讀者請參考相應的樹算法。本節內容旨在說明 VimL 的字典用法，展示其表達能力。而算法其實是與語言無關的。

在上述的樹結點字典結構中，只能從一個結點訪問其子結點，而無法從子結點訪問父結點。如果有這個需求，只要在每個結點字典中再加一個鍵引用父結點即可，如：

```
: let node.parent = {}
```

每個子結點都有父結點，即 parent 鍵非空。根結點沒有父結點，那 parent 鍵應該存個什麼值呢？可以就用空字典表示，也可以引用它自身，這都可以將根結點與其他非根結點區分開來。

我們知道，字典或列表變量都只是某個實體的引用。VimL 的自動垃圾回收機制主要是基於計數引用的。如果某個字典或列表實體沒有被任何變量引用了，即引用計數爲 0 時，(在變量離開作用域或顯式 ':unlet' 時會減少引用計數) VimL 就認定該實體無法被訪問了，就會當作垃圾回收其所佔用的內存。在大部分簡單場合中，這套機制很好用。不過考慮這裏討論的包含 parent 與 left right 鍵的樹結點，在父、子結點之間形成了環引用，它們的引用計數始終不會降到 0 。然而 VimL 另外也有一個算法檢測環引用，所以也儘可放心使用這個樹結構，不必擔心內存泄漏。只不過存在環引用時，垃圾回收的時機可能相對滯後而已。

現在，讓我們再考慮一種有任意多個孩子的樹（任意叉樹）。這種結構在實際應用中是存在的，比如目錄樹，每個目錄（結點）可以有很多個不確實數量的子目錄或文件（葉結點）。爲表示這種結構，我們可以將所有子結點放在一個列表中，然後用一個鍵引用這個列表，如下定義每個結點的字典結構：

```
: let node = {}
: let node.value = 0
: let node.parent = {}
: let node.child = []
```

與原來的二叉樹相比，取消 left 與 right 鍵，而以統一的 child 鍵代替。每當增加一個子結點時，就添加到 child 列表中，同時維護該子結點的 parent 鍵。如果 child 鍵爲空列表，就表示該結點爲葉結點。

圖

圖是一些頂點與邊的集合，常用 G(V, E) 表示，其中 V 是頂點集合，E 是邊集合，每條邊連接着 V 中兩個頂點。一般用 |V| 表示頂點的個數，|E| 表示邊數。

用程序表示圖，有兩種常用的方式，鄰接矩陣與鄰接表。這裏討論一下如何用 VimL 的數據結構表示圖。

鄰接矩陣很簡單，就是一個 |V| x |V| 大小的矩陣，假設就用變量名 graph 表示這個矩陣。前面小節已介紹，矩陣在 VimL 中就是列表的列表。如果頂點 i 與 j 之間有一條邊，就 ':let graph[i][j] = 1'，否則就用一個特殊值來表示這兩個頂點之間沒有邊，比如在很多情況下用 0 表示無邊是可行的，':let grapsh[i][j] = 0'。如果是有權邊，則可把邊的權重保存在相應的矩陣位置中，如 ':let graph[i][j] = w'。如果是無向圖，則再對稱賦值 ':let graph[j][i] = graph[i][j]'。

由於矩陣元素支持隨機訪問，用鄰接矩陣表示圖在某些應用中非常高效簡便，尤其在邊數非常稠密的情況下（極限情況是每兩個頂點之間都有邊的完全連通圖）。不過在邊數很少的情況下，這將是個稀疏矩陣，在內存空間使用上比較低效。

鄰接表，首先它是包含所有頂點的列表，每個頂點是一個字典結構，它至少有個鍵 edge 來保存所有與本頂點相關的邊，這應是一個邊結構的列表。在邊結構字典中則保存着權重 weight，以及它所連接的頂點（字典引用）。大致結構如下所示：

```
: let graph = [] " a list of vertex
: let vertex = {'edge': [], 'id':0, 'data': {}}
: let edge = {'weight':1, 'target': {}, 'source': {}}
```

如果只要求自上而下訪問邊結構，那這個字典中可以只保存一個頂點，另一個頂點就是它被保存的頂點（由它的 edge 鍵訪問到這個邊）。這可以減少一些存儲空間，不過頂點也只是字典引用，保存雙端點也浪費不了太多空間。

在實際的圖應用中，肯定還會有具體的業務數據，這些數據一般是保存頂點結構中。比如可以給每個頂點給個 id 編號或名字，如果有大量複雜的數據，可單獨保存在另一個字典引用中。

所以，鄰接表雖然複雜，但靈活度高，易擴展業務數據。而鄰接矩陣在矩陣元素中只能保存一個值，擴展有些不方便。除非是業務數據是保存在邊結構中，那麼在矩陣中可以保存另一個字典引用，而不是簡單的權重數值。

JSON

如果你瞭解 JSON，就會發現 VimL 的列表與字典的語法表示，正好也是符合 JSON 標准的。一個有效的 JSON 字符串也是合適的 VimL 的表達式，可以直接用於 ':let' 命令的賦值。

當然這有一個小小的限制，JSON 字符串不能有換行，因爲 VimL 語言是按行解析的，且續行符比較特殊（在下一行開頭使用反斜槓）。如果是不太複雜的 JSON，在 Vim 編輯中

可以將普通命令 J 將多行字符串合併爲一行，我不認爲你會用其他編輯器寫 VimL 腳本。

此外，有個內置函數 jsondecode() 可將一個合法的 JSON 字符串（允許多行）解析爲 VimL 值，以及反函數 jsonencode() 將一個 VimL 表達式轉換爲 JSON 字符串。

總結

在 VimL 中，用列表與字典的組合，可以表達很複雜很精妙的數據結構，幾乎只有想不到沒有做不到。其實這也不必奇怪，因爲目前大部分作爲高級語言的動態腳本，其思想是相通的。雖然 VimL 似乎只能用於 Vim，但它與其他流行的外部腳本語言，在某種程序上是極其相似的。

4.4 * 正則表達式

在本章末尾，再簡要介紹一下正則表達式的內容。正則表達式對於 Vim 很重要，但本教程不打處專門用一章的內容來講敘正則表達式（實際上正則表達式的內容可以寫一本書）。插錄在這章數據結構之後，你可以認爲正則表達式也是一種表達字符串內部結構的模糊方法——模糊其實比精確更難理解與掌握。

Vim 對於正則表達式的內置幫助文件請查閱 ':h pattern.txt'。

Vim 正則表達式的設置模式

很多編程語言或工具軟件，都支持正則表達式，所以這是一種很實用的通用技能。然而不幸的是各家支持的正則表達式都 "略有不同"，更不幸的是 Vim 自家裏面還有幾種不同的正則表達式語法，這是通過選項設置 &magic 改變正則表達式 "包裝套餐" 的。

Vim 一共支持四套正則表達式，在 / 或? 命令行中可添加特殊前導字符來表示本次搜索採用哪套正則表達式：

• \v(very magic)，最接近 perl 語言的正則表達式。除了常規標識符字符外，大多數字符都有特殊含義，即魔法字符。

• \m(magic)，這是 Vim 的標准正則表達式。主要特徵是括號與加號都是字面意義，不是魔法字符，需要在前面多加一個反斜槓來表示魔法意義。

• \M(nomagic) 更少的魔法字符，點號 (.) 與星號 (*) 都是字面意義。

• \V(very nomgic) 只有斜槓本身及正則表達式定界符有特殊意義，其他所有字符按字面意義匹配。

這四種正式表達式是根據魔法字符的多寡程度劃分的。但是用反斜框可以改變魔法字符的意義。即在一種正則表達式中，如果一個字符是魔法字符，反斜槓轉義後就表示字面意義；反之如果一個字符不是魔法字符，加反斜槓轉義後就可能成爲魔法字符表示特殊意義。例如在 \m 正則表達式中，加號 + 不是魔法字符，它匹配字面的加號，使用 \+ 表示匹配前面那個字符一次以上。而在 \v 正則表達式中，+ 是魔法字符，表示匹配前面那個字符一次以上，而用 \+ 匹配字面上的加號。

如果沒有顯式指定哪種正則表達式（這應該是大部分 vimer 使用 / 搜索的默認方式），就根據 &magic 選項決定。設定了‘:set magic’就默認使用 \m 正則表達式，設定‘:set nomagic’就默認使用 \M 正則表達式。但是若要使用 \v 或 \V 必須顯示指定。因爲 &magic 選項的默認值是開啓的，所以 Vim 的默認正則表達式是 \m 這套，不妨稱之爲 Vim 的標准正則表達式。

爲什麼正則表達式已經很複雜了，Vim 還要增加幾種非標准正則表達式來使之更復雜？我想這是 Vim 的另一個設計原則：儘可能減少用戶的手動輸入字符數（擊鍵次數）。Vim 是一個通用文本編輯器，所編輯的文件內容在不同場合或有不同的側重。比如在編輯程序源文件時，應該普遍會有很多括號，很可能就需要經常搜索字面意義的括號，這時用標准的 \m 正則表達式就更方便，而用 \v perl 類的正則表達式，就必須用 來搜索一對空括號。而在另外一些場合，可能希望直接用 () 來表示組合，這就用 \v 正則表達式更方便了。

對於精通（或習慣）perl 類正則表達式的用戶，可以通過簡單映射‘:nnoremap / /\v’自動添加 \v 前綴，始終使用 perl 類正則表達來搜索。在替換命令:s/// 的模式部分，也可以添加 \v 或其他前綴顯式指定正則表達式的標准。

但是，對於一些需要正則表達式作爲參數的內置函數，如 match()，只使用 magic 的標准正則表達式。這可能主要是考慮函數實現的方便與效率吧。畢竟函數主要寫在 VimL 腳本中，而腳本一般只需寫一次，語義一致也更重要。

因此，對於 Vimer 用戶，還是建議掌握 Vim 的標准正則表達式。對於其他三種非標准正則表達式，瞭解就好，覺得方便有用時，儘管一試。本文剩余部分只介紹 Vim 標准正則表達式的基本語法。

Vim 標准正則表達式

正則表達式描述的是如何匹配一個字符串，簡單地說，它試圖說明以下幾個基本問題：

- 匹配什麼字符
- 匹配多少次
- 在哪裏匹配（定位限制）

再高級的議題還有分組與前向自引用等。

匹配字面字符

以下字符按字面意義匹配（非魔法字符）：

- 大小字符與小寫字母：A-Z a-z
- 數字：0-9
- 下劃線：_
- 加號：+
- 豎線：|
- 小括號與大括號：()

• 其他沒有定義特殊意義的符號，以及其他指明要加反斜槓轉義才表示特殊意義的字符。

匹配字符類別

支持的常用字符類別有：

• \s：空白字符
• \d：數字字符（0-9）
• \w：單詞字符（合法標識符）
• \h：合法標識符的開頭
• \a：字母
• \l：小寫字母
• \u：大寫字母

以上這幾類字符表示，若改用大寫，則表示取反，如 \S 表示非空白字符。這與大多數正則表達式的語法表示是一致的。

Vim 正則表達式還有幾類字符表示與選項相關，由相應選項指定字符集。

• \i 由選項 &isident 指定的標識符
• \k 由選項 &iskeword 指定的關鍵字符
• \f 由選項 &isfname 指定的可用於文件名（路徑）的字符
• \p 由選項 &isprint 指定的可打印字符
• \I \K，\F，\P 大寫版本在以上小寫版本基礎上排除數字

也可以手動指定字符範圍，用中括號 []：

• [] 匹配括號內任意一個字符，如 [abcXYZ] 可匹配這六個字符中的任一個
• [0-9] 用短橫線（減號）指定的連續字符範圍，匹配該範圍內任一字符
• [^0-9] 匹配非數字，括號內第一字符是 ^ 時表示取反
• [-a-z] 若要包含減號本身，放在中括號內第一個字符，該示例表示小寫字母或減號

中括號的用法與其他多數正則表達式一樣。所以中括號與大小括號不一樣，它是魔法字符，若要匹配字面的中括號，則須用 [或]。

Vim 正則表達式還支持另一種特殊的中括號用法：

• \%[] 匹配中括號內可選的連續字符串，類似 ex 命令的縮寫語法。

例如‘:edit’可縮寫至‘:e’，用正則表示就是‘:e\%[dit]’。

其他一些特殊字符：• . 任一單字符
• \t 製表符
• \n 換行符
• \r 回車符
• \e <Esc> 鍵

匹配重複多次

- 0 或多次：*
- 1 或多次：\+
- 0 或 1 次：\? 或 \=
- 指定次數範圍：\{n,m}，右大括號不需要反斜槓轉義，左大括號需轉義
- 非貪婪的次數範圍：\{-n,m}

所以，沒有意外地，點號與星號是魔法字符，分別用於匹配任意字符與任意次數，加反斜槓匹配字面點號（\.）與星號（*）。但是問號? 不是魔法字符，須用 \? 來表示匹配 0 或 1 次。

這些語法項不能單獨使用，須用於表示字符（或類別）的後面，表示匹配前面那個字符字符多少次。\{n,m} 是通用的次數表示語法，可以省略 n 或（與）m。

- \{n} 嚴格匹配 n 次
- \{n,} 至少要匹配 n 次
- \{,m} 匹配 0 至 m 次
- \{} 匹配 0 或多次，等同於 *
- \{0,1} 匹配 0 或 1 次，等同於 \?
- \{1,} 匹配 1 或多次，等同於 \+

正則表達式一般採用貪婪算法，以上的 \{n,m} 及 * \+ 都是儘可能匹配更多次。在一些場合需求下，需要採用非貪婪算法，可用 \{-n,m} 表示儘可能匹配更少次數。\{-n,m} 也有省略變種，與 \{n,m} 用法一樣，只是在左大括號內開始多一個減號。

例如，對於字符串 hello world!，如果用正則表達式 .* 或 \.\{} 就能匹配整個字符串，因爲是在 標籤之間貪婪匹配儘可能多的字符。但如果是 \.\- 則只能匹配前一個標籤，即子字符串 hello，這就是非貪婪的意義。（注意：如果在 Vim 中測試該例，在 / 命令行輸入這些正則表達式，須注意轉義 / 本身，即應該輸入 /.*<\/b>）

匹配定界符（錨點）

- ^ 匹配行首
- $ 匹配行尾
- < 匹配詞首
- > 匹配詞尾

在 Vim 編輯過程中，按 * 或 # 命令，用於搜索當前光標下的單詞，就會在當前單詞前後自動加上 \< 與 \!> 表示界定匹配整個單詞。例如，你將光標移到本文的 hello 單詞上，按下 *，Vim 應該會高亮所有 hello 單詞，但如果有個地方寫成 hello_world 加了下劃線連字符，那就不會高亮這裏的 hello 前綴。使用 ':reg /' 可以查看 Vim 爲我們自動添加的正則表達式爲 \<hello\>，你也可以按 / 進入搜索命令後再按向上方向鍵把上次的搜索模式複製到當前命令行中查看。

- \zs 不匹配任何東西，只標定匹配結果的開始部分
- \ze 不匹配任何東西，只標定匹配結果的結束部分

這兩個標記不影響“是否匹配”的判斷，隻影響若匹配成功後實際匹配的結果子字符串。例如先看個簡單模式 hello.*world!，它可以匹配 hello world! 或者在這兩個單詞之間添加了其他亂七八糟的字符後也能匹配，匹配結果是從 hello 到 world! 之間的所有長字符串。但是另一個類似模式\zshello\ze.*world!，它與前面那個模式能匹配一樣的字符串或文本行，但是匹配結果只有前面那個 hello 單詞而已。可以利用 Vim 搜索的高亮顯示來理解這個差異。

所以如果僅爲了搜索，\zs 與 \ze 是基本不影響結果的，但如果同時要替換時，這兩個標定就很有用了，可使替換命令或函數大爲簡化。

例如將 hello 改爲首單詞大寫 ‘:s/\zshello\ze.*world!/Hello/’，它只會修改後面還接了 world! 的 hello，單獨這個單詞卻不會被修改的。

Vim 的正則表達式，還有另外一些定位擴展，以 \% 形頭的：

- \%^ 匹配文件開頭
- \%$ 匹配文件結束
- \%l 匹配行，在 \% 與 l 之間應該是一個有效的數字行號，表示匹配相應的行，若在行號前再加個 < 表示匹配該行之前的行，加個 > 則表示匹配之後的行。例
- 如 \%23l，\%<23l，\%>23l 等。
- \%c 匹配列，與 \%l 用法類似。
- \%# 匹配當前光標位置
- \%’m 匹配標記 m，m 可以是任一個命令標記（mark）。

分組與引用

- \(\) 創建一個分組（子表達式），它本身不影響匹配，但便於其他語法功能使用
- \1 \2 ... \9 依次引用前面用 \(\) 創建的分組
- \%(\) 多加一個% 與 \(\) 創建分組一樣功能，但又不當作一個子表達式，即不影響 \1、\2 等的引用次序。

分組的子表達式引用也可用於替換命令的替換部分，這可能是前向引用的常用用法，例如”:s/\(\d+\) + \(\d\+\)/\2 + \1/” 用於將一個加法運算的表達式的兩個操作數調換次序，將 123 + 321 修改爲 321 + 123。

當然在正則表達內部也能用前向引用以達到某些特殊要求，比如常見的匹配 html 配對標籤，<\(.*\)>hello<\/\1> 可以匹配用任意標籤括起的 hello 單詞，如 hello、<xyz>hello</xyz> 等，但若標籤不配對不能匹配。

其他限定語法

- \c 忽略大小寫
- \C 不能忽略大小寫

在默認情況下，正則表達式匹配也受 &ignorecase 的影響，但如果在一個模式中任意地方加上了 \c 或 \C 控制符，就強行忽略或不忽略大小寫。一般是加在表達式末尾，臨時改變主意在怎麼忽略大小寫。

這與 \m 或 \M 的控制符不一樣，它隻影響後續正則表達式的魔法字符釋義。不過建議放在整個正則表達式最前面爲好。

Vim 正則表達式總體構成

正則表達的具體語法細節，需要經常翻手冊確認。不過最後還是再歸納一下 Vim 正則表達式的總體構成定義，按幫助文檔的術語，一個正則表達式從上到下分以下幾個層次：pattern <– branch <– concat <– piece <– atom <– item 。

1. 一個正則表達式也叫一個模式（pattern），一個模式可能由多個分支（branch）構成，雖然大多應用場合下只有一個分支。多個分支由 \| 分隔，表示“或”的語義（| 不是魔法字符，所以要用 \|）。任一個分支匹配目標字符串，則表示該模式匹配成功；如果多個分支都匹配，則匹配結果取第一個能匹配的分支。

2. 每個分支可能由一個或多個聚合（concat）組成，若多個聚合由 & 分隔，這表示“且”的語義。必須匹配每一部分的聚合，該分支纔算匹配，但匹配結果是按最後一個聚合的匹配爲准。

3. 每個聚合又可以由多個分子（piece）構成，分子之間相當於有自然引力結合，無須特殊字符直接粘接。如模式 abc 就只一個分支，一個聚合，該聚合有三個分子，每個分子是簡單的字面字符；模式 a[0-9]c 或 a\dc 同樣是由三個分子組成。

4. 每個分子又由一個或多個原子（atom）構成。上一小節講述的正則表達式語法其實主要都處於這一層。字符類別如 \d，\s，\w 就是表示一個原子，用 [] 指定的字符集合，也只是一個原子，而表示重複次數的 \{n,m} 就是描述多個原子的情況。

5. 每個原子即可以是普通原子，又可以是循環定義的子模式，即用 \(\) 或 \%(\) 創建的分組。

以上的第 1 第 2 層相當於邏輯或與邏輯且用於正則表達式的上層擴展，第 3 層卻是很平凡的定義，語法細節最多的是第 4 層，而第 5 層則是更深入的高級用法。

小結

正則表達式是很精妙的技術，非短時間所能掌握，只有多加實踐積累經驗。在 Vim 中，可多利用高亮模式（set hlsearch）來測試正則表達式的正確性。有其他語言或工具的正則表達式經驗的用戶，則特別注意一下 Vim 的特性語法。

正則表達的匹配是很複雜的算法，在其他一些語言中，可能有預編譯正則表達式的功能（庫函數）。但在 VimL 中，似乎還未提供類似的內建函數。不過在寫較大的 VimL 腳本時，如果涉及使用正則表達式，也建議將常用的正則表式（字符串）統一定義在腳本開頭，方便管理與修改。

第五章　VimL 函數進階

在第二章中，我們已經講敘了基本的函數定義與調用方法，以及一些函數屬性的作用。但正如大多數編程語言一樣，函數是如此普遍且重要的元素。因而本章繼續討論一些有關函數的較爲高級的用法。

5.1　可變參數

可變參數的意義

一般情況下，在定義函數時指定形參，在調用函數時傳入實參，且參數個數必須要與定義時指定的參數數量相等。但在一些情況下，我們將要實現的函數功能，它的參數個數可能是不確定的，或者有些參數是可選的，可缺省使用默認值。這時，在函數定義中引入可變參數就非常方便了。相對於可變參數，常規的形參也就是命名參數。

• 在函數頭中，用三個點號 '...' 表示可變參數，可變參數必須用於最後一個形參，如果有其他命名參數，則必須位於 '...' 之前。

• 在函數體中，分別用 a:1，a:2 ……等表示第一個、第二個可變參數。用 a:0 表示可變參數的數量，a:000 是由所有可變參數組成的列表變量。

• 命名參數最多允許 20 個，雖然大部分情況也夠用了。可變參數的數量沒有明確限制。

• 調用函數時，傳入的實參數量至少不低於命名參數的數量，但傳入的可變參數數量可以爲 0 或多個。當沒有傳入可變參數時，a:0 的值爲 0。

需要強調的是，只有定義了 '...' 可變參數，才能在函數體中使用 a:0，a:000，a:1 等特殊變量。較好的實踐是先用 a:0 判斷可變參數個數，然後視情況使用 a:1，a:2 等每個可變參數。如果只傳入一個實參，卻使用了 a:2 變量，會發生運行時錯誤。此外 a:000 就當作普通列表變量使用好了，a:000[0] 就是 a:1，因爲列表元素索引從 0 開始。

例如，可用以下函數展示可變參數的使用方法：

```
1 function! UseVarargin(named, ...)
2     echo 'named argin: ' . string(a:named)
3 
4     if a:0 >= 1
5         echo 'first varargin: ' . string(a:1)
6     endif
```

```
7       if a:0 >= 2
8           echo 'second varargin: ' . string(a:2)
9       endif
10
11      echo 'have varargin: ' . a:0
12      for l:arg in a:000
13          echo 'iterate varargin: ' . string(l:arg)
14      endfor
15  endfunction
```

你可以用‘:call’調用這個函數，嘗試傳入不同的參數，觀察其輸出。可見有兩種寫法獲取某個可變參數，比如用 a:1 或 a:000[0]，視業務具體情況用哪種更方便。而且 a:000 還可用列表迭代方法獲取每個可變參數。

不定參數示例

在 2.4 節，我們已經定義了一個演示之用的函數 Sum 可計算兩個數之和，簡化重新截錄於下：

```
1  function! Sum(x, y)
2      let l:sum = a:x + a:y
3      return l:sum
4  endfunction
```

現假設要計算任意個數之和，則可改爲如下定義：

```
1  function! Sum(x, y, ...)
2      let l:sum = a:x + a:y
3      for l:arg in a:000
4          let l:sum += l:arg
5      endfor
6      return l:sum
7  endfunction
```

這裏認爲調用 Sum() 時必須提供兩個參數，否則求和沒有意義。其實也可以定義爲 Sum(...)，將函數實現中的 l:sum 初始化爲 0 即可。

若一個函數用 Fun(...) 定義，只聲明瞭可變參數，則可用任意個參數調用，非常通用。然而過於通用也表明意義不明確，良好的實踐是，除非有必要，儘可能用命名參數，少用可變參數。使用合適的參數變量名，函數的可讀性增強，使用可變參數時，最好加以註釋；同時也建議在函數前面部分判斷可變參數數量與類型，第一時間分別賦於另外的局部變量，也能增加函數的可讀性。

調用這個求和函數時，用‘:call Sum(1, 2, 3, 4)’方式。事實上，只爲這個需求的話，不必用可變參數，直接用一個列表變量作爲參數可能更方便。如改寫爲:

```
function! SumA(args)
    let l:sum = 0
    for l:arg in a:args
        let l:sum += l:arg
    endfor
    return l:sum
endfunction
```

這個函數的意義是爲一個列表變量內所有元素求和，以‘:call Sum([1, 2, 3, 4])’方式調用。然而需要注意的是，並非所有用可變參數的函數，都適合將可變參數改爲一個列表變量。

默認參數示例

在 VimL 的內置函數中，格式化字符串的 printf() 就是接收任意個參數的例子。另外還有大量內置函數是支持默認參數的，如將列表所有元素連接成一個字符串的 join()。這種情況與不定參數略有不同，它能接收的有效參數個數是確實的，只是在調用時後面一個或幾個參數可以省略不傳，不傳實參的話就自動採用了某個默認值而已。

比如我們也可以自己實現一個類似的函數 Join():

```
function! Join(list, ...)
    if a:0 > 0
        let l:sep = a:1
    else
        let l:sep = ','
    endif
    return join(a:list, l:sep)
endfunction
```

雖然可以（更低效率）用循環連接字符串，但這是爲簡要說明原理，直接調用內置的 join() 完成實際工作了。關鍵點是提供了另一個逗號作爲默認分隔字符，通過 a:0 來判斷傳入的可變參數個數，再給分隔字符賦以合適的初始值。其實這個 if 分支可以直接用 get() 函數代替：let l:sep = get(a:000, 0, ',')。這用起來更爲簡潔，不過用 if 分支明確寫出來，更容易擴充其他邏輯，即使是用 echo 打印個簡單的日誌。

間接調用含可變參數的函數

一般情況下，函數都不是獨立完成工作的，往往還需要調用其他的函數。假如一個支持可變參數的函數內，要調用另一個支持可變參數的函數，給後者傳遞的參數依賴於前者

接收的不確定的參數，這情況就似乎變得複雜了。

爲說明這種應用場景，先參照上述 Sum() 函數再定義一個類似的連乘函數：

```
function! Prod(x, y, ...)
    let l:prod = a:x * a:y
    for l:arg in a:000
        let l:prod = l:prod * l:arg
    endfor
    return l:prod
endfunction
```

注：VimL 支持‘+=’操作符，卻不支持‘*=’操作符，請參閱 ‘:h +=’。

然後再定義一個更上層的函數，根據一個參數分發調用連加 Sum() 或連乘 Prod() 函數，傳入剩余的不定參數：

```
function! Calculate(operator, ...)
    echo Join(a:000, a:operator)
    if a:operator ==# '+'
        " let l:result = Sum(...)
        " let l:result = Sum(a:000)
    elseif a:operator ==# '*'
        " let l:result = Prod(...)
        " let l:result = Prod(a:000)
    endif
    return l:result
endfunction

echo Calculate('+', 1, 2, 3, 4)
echo Calculate('*', 1, 2, 3, 4)
```

在這個示例函數中，第一行的 echo 語句用於調試打印，不論是用剛纔自定義的 Join() 或內置的 join() 函數都能正常工作。但是在隨後的 if 分支中，不論是 Sum(...) 還是 Sum(a:000) 都不能達到預期效果，雖然它作爲“僞代碼”很好地表達了使用意途，所以先將其註釋了。

先分析原因，Sum(...) 是語法錯誤。因爲‘...’只能用於函數頭表示不定參數，卻不能在函數體中表示接收的所有不定參數。a:000 可以表示所有不定參數，但它只是一個列表變量，調用 Sum(a:0000) 時只傳了一個參數變量，而原來定義的 Sum() 函數要求至少兩個參數，所以也會出錯誤，因爲相當於調用 Sum([1,2,3,4]) 也是錯誤的。

解決辦法是用 call() 函數間接調用，它的第一個參數是一個函數，第二個參數正是一個列表，這個列表內的所有元素將傳入第一個參數所代表的函數進行調用。例如，這語句

“:echo call(‘Sum’, [1,2,3,4])” 能正常工作。於是可將 Calculate() 函數改寫：

```
function! Calculate(operator, ...)
    if a:0 < 2
        echoerr 'expect at leat 2 operand'
        return
    endif

    echo Join(a:000, a:operator)
    if a:operator ==# '+'
        let l:result = call('Sum', a:000)
    elseif a:operator ==# '*'
        let l:result = call('Prod', a:000)
    endif

    return l:result
endfunction
```

這裏再作了另一個優化，先對不定參數個數作了判斷，不足兩個時則返回錯誤。

5.2 函數引用

關於函數引用的幫助文檔先給傳送門‘:h FuncRef’（注意大寫）。很多函數的高級用法都在函數引用基礎上建立的。

函數引用的意義

繼續接着上一節的內容引申來講。例如在 Calculate() 函數中間接調用 Sum() 時須用如下語法：call('Sum', a:000) ，'Sum' 函數名須用引號括起來當作一個字符串參數傳入。

如果嘗試執行‘:echo call(Sum, [1,2,3,4])’就會報 E121 的“未定義變量”錯誤。也就是說，Sum 是一個自己定義的函數，但函數與變量在 VimL 中有本質的不同，而 call() 要求一個變量作爲參數，所以不能直接將函數傳入。然而這個變量又要求能代表函數，所以 VimL 就需要一個“函數引用”的概念。

就這個特殊的 call() 而言，在第一參數中將一個函數名用引號括起的字符串也能達到引用一個函數的目的，但這顯然是不正式不通用的。函數引用也是一個變量，不過是另一種特殊的變量（值）類型，不應該與簡單的字符串變量類型混淆。

在其他一些編程（腳本）語言中，函數是所謂的一等公民，即與變量的地位一樣，可以用變量的地方，也可以用函數。但在 VimL 設計之初，函數與變量就是兩個不同次元的東西。只有在引入了函數引用之後，函數引用與變量纔是相同的東西。

函數引用的定義

可以內置函數 function() 創建一個函數引用，其參數就是所要引用的函數名（引號字符串），既可以是內置函數也可以是自定義函數的名字。例如：

```
: let Fnr_Sum = function('Sum')
: echo type(Fnr_Sum)
: echo Fnr_Sum(1,2,3,4)
```

上例創建一個變量 Fnr_Sum，它引用自定義函數 Sum()。查看這個變量的類型，顯示是 2，這就是函數引用的類型（v:t_func）。然後這個函數引用可以像引用的那個函數一樣調用，也就是後面接括號傳入參數列表。

函數引用變量與函數本身的關係，就與之前所述的列表（或字典）變量與列表（或字典）實體之間的關係。在常規運用場合中，一般可不必理會其中的差異，凡是要求函數調用的地方，都可以用函數引用代替。而且，函數引用作爲一個變量，使用範圍將更加靈活。因爲 VimL 的變量是弱類型的，在使用變量時不檢查變量類型，所以在任何使用變量的地方，也都可以使用函數引用代替。當然，你不能試圖對函數引用進行加減乘除這樣的操作，那會觸發運行時錯誤，函數引用主要（也許是唯一）支持的操作就是調用。

VimL 的變量名自有其規則（見第二章），而函數引用的變量名在此規則上還有更嚴格一點的限制，就是必須也以大寫字母開頭。這是因爲要與函數名的規則吻合。因爲從代碼語法上看一個函數調用，無從分辨它是函數引用還是函數本身。主要注意如下幾點：

- 函數引用變量也可以加作用域前綴，如果加了 ‘s:’, ‘w:’, ‘t:’ 或 ‘t:’ 這幾個前綴，則不再要求變量名主體以大寫字母開始了，因爲這種情況下不會有歧義。參數作用域前綴 ‘a:’ 用於函數引用之前，也不必大寫字母。
- 如果在函數引用變量名之前加全局作用域前綴 ‘g:’ 或局部作用域前綴 ‘l:’，仍然要求其變量名主體以大寫字母開頭。因爲這兩種前綴是可以省略的，要保證省略後的等價的“裸”調用仍然合乎函數調用規則。
- 函數引用變量名，不能與已有的自定義函數名相同，否則也會發生歧義，vim 將無從分辨是觸發調用函數引用呢，還是觸發調用同名函數本身。
- 函數引用變量名允許與已存在的其他變量名重名，只不過其含義是重定義或覆蓋原變量的意義，雖然語法上合法，但不建議這麼做。

再次提醒一下，function 這個 “關鍵字”，既是一個命令名，也是一個內置函數名。用 ‘:function 命令是創建或定義一個函數，而 function() 函數則是創建或定義一個函數引用（其參數須是已由 ‘:function’ 命令創建的函數名，或內置函數名）。命令與函數是完全不同空間次元的東西，也與變量互不相關。如果你願意，甚至也可以自定義一個叫 function 的變量，但最好不要這樣做。

在 VimL 中，有很多內置函數與命令重名，用於實現相似的功能。上節剛用到過的 call() 函數與 ‘:call’ 命令也是這種情況。在查 vim 幫助文檔時，查函數時在後面加對空括號，查命令時在前面加個冒號。另一方面，VimL 的內置變量名都是以 ‘v:’ 前綴的，這倒不必擔心混淆。

函數引用的使用

下面再講解函數引用的使用建議與示例。仍以上節末用於實現不定參數連加或連乘的 Calculate() 函數爲例。

將函數引用作爲參數傳遞

首先，不建議使用全局的函數引用變量。因爲用 ':function' 命令定義的函數是全局的，儘量不要將函數引用也定義在全局作用域中，避免麻煩。例如，可將上節的 Calculate() 函數改爲如下使用函數引用的方式（爲簡便起見，略過參數檢測）：

```
function! CalculateR(operator, ...)
    if a:operator ==# '+'
        let l:Fnr = function('Sum')
    elseif a:operator ==# '*'
        let l:Fnr = function('Prod')
    endif

    let l:result = call(l:Fnr, a:000)
    return l:result
endfunction
```

這裏先根據參數創建一個函數引用 Fnr，在函數內定義的變量都是局部變量，'l:' 前綴可選。然後這個函數引用也可以作爲參數傳給 call() 函數，它能同時處理作爲函數名的字符串變量類型或函數引用類型，反正都是用以訪問實際所調函數的手段；也不妨認爲在之前傳入字符串時，call() 函數也會自動先調用 function() 獲得函數引用。

腳本局部函數及引用

上面改寫的 CalculateR() 函數有一處不太好，就是每次調用都要重新創建 l:Fnr 這個相同的函數引用變量，略顯低效。在實踐中，函數定義一般是寫在單獨的腳本中，因此函數引用也可以定義爲 's:' 腳本局部變量。例如：

```
" File: ~/.vim/vimllearn/funcref.vim

let s:fnrSum = function('Sum')
let s:fnrProd = function('Prod')

function! CalculateRs(operator, ...)
    if a:operator ==# '+'
        let l:Fnr = s:fnrSum
    elseif a:operator ==# '*'
```

```
10          let l:Fnr = s:fnrProd
11      endif
12
13      let l:result = call(l:Fnr, a:000)
14      return l:result
15  endfunction
```

注意，如前所述，'s:' 前綴的函數引用變量可用小寫開頭，'l:' 或缺省前綴的函數引用須大寫開頭。這裏主要爲演示不同前綴的函數函數引用變量，其實 l:Fnr 中間變量也可省去，直接將 call() 調用語用寫在 if 分支中。

這樣，s:fnrSum 與 s:fnrProd 函數（引用）就是私有的了，只能在該腳本內使用，而 CalculateRs() 函數仍定義爲全局函數，提供爲外部公用接口。但是，那兩個私有變量引用的仍是公用的函數 Sum() 與 Prod()。如果想再要隱藏，可以將這兩個函數也定義爲 's:' 的作用域：

```
1  " File: ~/.vim/vimllearn/funcref.vim
2
3  function! s:sum(...)
4      let l:sum = 0
5      for l:arg in a:000
6          let l:sum += l:arg
7      endfor
8      return l:sum
9  endfunction
10
11 function! s:prod(...)
12     let l:prod = 1
13     for l:arg in a:000
14         let l:prod = l:prod * l:arg
15     endfor
16     return l:prod
17 endfunction
18
19 let s:fnrSum = function('s:sum')
20 let s:fnrProd = function('s:prod')
21
22 echo s:
```

這裏的 s:sum() 函數對比原 Sum() 略有修改，不再強制要求至少兩個參數。同時函數

名加上 's:' 前綴後，也不再強制要求以大寫字母開頭。當用 function() 創建函數引用時，須將 's:sum' 整個字符串當作該腳本局部數字的 "名字" 傳入爲參數。

然後，重點迷惑來了，腳本內的 s:sum() 實際函數名其實並不是 's:sum'！這只是語法上規定的書寫文法。在 vim 內部，會將 's:' 前綴的函數名替換爲 <SNR> 編號 _。其中編號是指 vim 在加載該文件時對其賦與的編號。可用 ':scriptnames' 命令查看當前 vim 所加載過的所有腳本，一般情況下編號爲 1 的第一個加載文件就是你的起始配置文件 vimrc，然後每次加載腳本時順序編號。所以 s:sum() 腳本私有函數的實際名字是動態變化的，在不同的 vim 會話中加載時機極可能不一樣，其編號中綴也就不一樣了。

如果在腳本末尾加上 'echo s:' 這個語句（'s:' 是一個特殊字典，保存着該腳本內定義的所有以 's:' 前綴開始的腳本局部變量），那麼在加載該腳本時，將回顯如下信息：

```
{'fnrSum': function('<SNR>77_sum'),
 'fnrProd': function('<SNR>77_prod')}
```

表明在這次 vim 會話環境中，s:sum() 函數名實際上是 <SNR>77_sum，也可以直接用這個名字來調用該函數，如在命令行中輸入

```
: echo <SNR>77_sum(1, 2, 3, 4)
```

是能正常工作中的。

因此，看似腳本局部私有的 s:sum() 實際上是被轉化成了 <SNR>77_sum() 全局公有函數。其中 <SNR>77_ 前綴在在某些地方也可用特殊符號 <SID> 表示。當然，任何正常的人，都不會採用後者來調用函數，況且腳本編號都是臨時賦與的不保存一致性，於是也算達到了作用域隱藏的目的。

另外，還有一點要注意的是，s:sum() 是函數，不是變量，所以它不會被保存在 's:' 字典內。只有函數引用變量 s:fnrSum 與 s:fnrProd 才保存在 's:' 字典內，其鍵就是變量名 fnrSum 與 fnrProd，其值就是相應的函數引用。顯然，vim 不能自作主張地自動爲 s:sum() 創建一個名爲 s:sum 的函數引用變量，甚至我們自己也不能手動用 ':let ... function()' 語句創建名爲 s:sum 的函數引用變量，否則在調用 s:sum(1,2,3,4) 是就會發生語法歧義。但是，我們能用它創建其他類型的變量，如在腳本末尾加入如下代碼並重新用 ':source' 加載：

```
1 " File: ~/.vim/vimllearn/funcref.vim
2
3 " let s:sum = function('s:sum') " 錯誤
4 " let s:prod = function('s:prod')
5
6 let s:sum = '1+2+3+4'
7 let s:prod = '1*2*3*4'
8 echo s:
9
```

```
10 echo s:sum(1,2,3,4)
11 echo s:prod(1,2,3,4)
```

可以把 s:sum 賦值爲字符串類型變量，然後 s:sum() 函數並未失去定義，仍然可正常調用。所以，'s:' 作用域前綴用於變量與函數前有着不同的實現意義。s:sum() 函數本質上是 <SNR>77_sum() 函數，與 s:sum 變量大有不同。然而，正常的程序猿非常不建議玩這樣的雜耍。

將函數引用收集在列表中

在前一示例中，在腳本中創建的 's:' 前綴的函數引用變量，被自動地收集保存在一個特殊字典中。這表明函數引用與普通變量"無差別"的同等地位，可以用在任何需要變量的地方。比如，我們也可以主動地將函數引用保存在一個列表中，以實現某些特殊功能：

```
1 " File: ~/.vim/vimllearn/funcref.vim
2
3 let s:operator = [function('s:sum'), function('s:prod')]
4 function! CalculateA(...)
5     for l:Operator in s:operator
6         let l:result = call(l:Operator, a:000)
7         echo l:result
8     endfor
9 endfunction
```

這裏，我們定義了一個列表變量 s:operator ，其元素都是能接收不定參數的運算函數的引用。然後在函數 CalculateA() 中遍歷該列表，爲每個函數傳遞參數進行計算。這是個全局函數，所以加載腳本後，可直接在命令行中執行 ':call CalculateA(1,2,3,4)' 驗看結果。

仍然要注意的是，在 for 循環中，循環變量 l:Operater 仍然要以大寫字母開頭，才能接收 s:operator 列表內的函數引用變量。否則，若以小寫字母的話，有可能省去 'l:' 前綴，寫出類似 operator(1,2,3,4) 的函數調用，這就有語法錯誤了，因爲小寫字母的函數名調用，都保留給 VimL 的內置函數。

良好的實踐是，始終以大寫字母開頭命名函數引用變量，不管什麼作用域前綴；如果不嫌麻煩，再以 Fnr 爲變量名前綴也未嘗不可。

5.3 字典函數

函數引用能保存在字典，這不意外，上節就提到過，腳本內定義的 's:' 前綴變量（包括函數引用），就自動保存在 's:' 這個特殊字典中。關鍵是如何主動利用這個特性，爲編程需求帶來便利。在本節中，將保存在字典中的函數引用簡稱爲字典函數。

將已有函數保存在字典中

沿用上節的示例，將函數引用保存在字典中，相關代碼改寫如下：

```
" >>File: ~/.vim/vimllearn/funcref.vim

let s:dOperator = {'desc': 'some function on varargins'}
let s:dOperator['+'] = function('s:sum')
let s:dOperator['*'] = function('s:prod')

function! CalculateD(operator, ...) abort
    let l:Fnr = s:dOperator[a:operator]
    let l:result = call(l:Fnr, a:000)
    return l:result
endfunction
```

這裏先定義了一個字典變量 s:dOperator，並用鍵 '+' 保存函數 s:sum() 的引用，用鍵 '*' 保存函數 s:prod() 的引用。然後改寫 CalculateD() 函數就很簡潔了，根據傳入的第一參數索引字典，獲得相應的函數引用，再調用之。因爲直接用鍵索引字典，且認爲沒有遍歷全部鍵的需求，所以還可以在 s:dOperator 字典加入非函數引用的鍵，比如 desc 保存了一條描述，字符串類型。

可以在命令行中輸入 ':echo CalculateD('*', 1, 2, 3, 4)' 測驗一下。注意到該函數沒有檢查傳入參數是否有效的鍵，如 ':echo CalculateD('**', 1, 2, 3, 4)' 會報錯。可以先用 has_key() 內置函數檢查參數 a:operator 是否存在的鍵，更進一步，可再用 type() 函數與該鍵相關聯的值是否函數引用。如果參數是非法的，則提前返回，至於返回什麼值表示錯誤，那就與具體需求有關了。也許在某些情況下，不檢查參數，直接讓它在出錯時終止腳本運行也是可接受的處理方式。

按成員的方式引用函數

我們知道，字典元素有兩種索引方式，一是用方括號（類似列表索引），一種是用點號（類似成員索引）。不過後者只是前者的語法糖，要求鍵名是簡單字符串（有效標誌符）。因此可以用一個較有意義單詞鍵名來代替 '+'，'*' 符號鍵名，例如：

```
" >>File: ~/.vim/vimllearn/funcref.vim

let s:dOperator.sumFnr = s:dOperator['+']
let s:dOperator.prodFnr = s:dOperator['*']
echo s:dOperator.sumFnr(1, 2, 3, 4)
echo s:dOperator.prodFnr(1, 2, 3, 4)
```

如果之前沒有在字典中定義 '+' 鍵，也可以直接用 let s:dOperator.sumFnr = function('s:sum') 獲得函數引用。這裏以小寫字母開頭的鍵名也可以保存函數引用。然後調用函數的寫法就是 s:dOperator.sumFnr()。由於使用的是腳本局部的字典變量，須用 ':source' 命令重新加載腳本文件執行上例，觀察這種調用方法的結果。

直接定義字典函數

爲了在字典鍵中保存一個函數引用，之前其實分了三步工作：

1. 用 ':function' 命令定義一個函數；
2. 用 ':function()' 函數獲取這個函數的引用；
3. 用 ':let' 命令將這個函數引用賦值給字典的某個鍵。

但這三步曲（實際是兩條語句）可以合起來，直接在定義函數時就將其引用保存在字典中，其語法示例如下：

```
" >>File: ~/.vim/vimllearn/funcref.vim

function s:dOperator.sum(...)
    let l:sum = 0
    for l:arg in a:000
        let l:sum += l:arg
    endfor
    return l:sum
endfunction

function! s:dOperator.prod(...)
    let l:prod = 1
    for l:arg in a:000
        let l:prod = l:prod * l:arg
    endfor
    return l:prod
endfunction

echo s:dOperator.sum(1, 2, 3, 4)
echo s:dOperator.prod(1, 2, 3, 4)
```

其實就相當於將之前的函數頭':function s:sum(...)'改爲':function s:dOperator.sum(...)'，函數體功能實現完全一樣。要注意的是在執行這一行之前，s:dOprator 字典必須是已定義的。然後調用該函數的用法完全一樣。

請注意區分一下，s:dOperator.sumFnr 顯然是一個函數引用，它引用事先已定義的 s:sum() 函數。s:dOperator.sum 也是一個函數引用，它引用的又是哪個函數呢？它引用的

是即時定義的函數，它沒有名字（沒機會也沒必要給個名字），也叫做匿名函數。在 Vim 內部，它將給這樣定義的匿名函數一個編號，所以也叫編號函數。

如果在腳本文件末尾寫上 'echo s:' 這條語句，根據其輸出結果，就能更清楚地分辨這些函數引用變量的異同。例如，執行結果大概相當於如下定義：

```
s:fnrSum = function('<SNR>77_sum')
s:fnrProd = function('<SNR>77_prod')

s:dOperator['+'] = function('<SNR>77_sum')
s:dOperator['*'] = function('<SNR>77_prod')

s:dOperator.sumFnr = function('<SNR>77_sum')
s:dOperator.prodFnr = function('<SNR>77_prod')

s:dOperator.sum = function('172')
s:dOperator.prod = function('173')
```

因此，s:fnrSum，s:dOperator['+'] 與 s:dOperator.sumFnr 都是引用同一個函數，那就是 s:sum() 局部函數，不過 vim 自動將其修正爲 <SNR>77_sum() 全局函數。而 s:dOperator.sum 則完全引用另一個函數，是編號爲 172 的匿名函數。當然，你的輸出中，腳本編號與函數編號極可能是不一樣的。

我們知道，退化的 ':function' 命令可以查看打印函數定義。所以可以用 ':function <SNR>77_sum' 在命令行直接執行，其輸出應該與腳本中定義的 s:sum() 函數一致。但是在命令行使用 ':function s:sum' 是錯誤。那匿名函數怎麼查看呢，直接用編號作爲參數是不行的，需用一個大括號括起來，如：

```
: function <SNR>77_sum
: function {173}
```

但是，用於獲取一個函數引用的 function() 卻無有效方法僅從匿名函數的編號獲得其引用。如 function('173') 或 function('173') 都不能正常工作。匿名函數一般必須在創建時賦值給某個函數引用變量，然後只能通過該函數引用調用之。當然了，該函數引用可以再賦值給其他變量就是。

字典函數的特殊屬性

如果仔細觀察上述 ':function 173' 命令輸出，可以發現它在函數頭定義行尾，自己添加了一個關鍵字 dict，表示將要定義的函數具有 dict 屬性。這個屬性指出該函數必須通過字典來激活調用，也就是說必須將其引用保存在字典的某個鍵中。然後在函數體中，可以使用 self 這個關鍵字，它表示調用該函數時所用到的字典變量。

例如，假設我們要在上述 s:dOperator 字典中另外加一個計算圓面積的函數。從數學

上講，圓面積只是其半徑的函數，應該只要傳入半徑參數。但在程序中實現計算時，還要涉及一個圓周率常量。這個常量不適合放在函數內定義，當然可以定義爲 's:' 腳本變量，不過最好還是保存在同一個字典中。

```
" >>File: ~/.vim/vimllearn/funcref.vim

let s:dOperator.PI = 3.14
function! s:dOperator.area(r)
    return self.PI * a:r * a:r
endfunction

echo s:dOperator.area(2)
```

我們先定義了 s:dOperator.area 函數（引用），然後調用 s:dOperator.area(2) 來計算半徑爲 2 的圓面積。在函數定義體內用到了 self.PI，這個 self 就是調用該函數時所用到的字典變量，也即 s:dOperator。

這裏，我們調用時與定義時用到的字典變量是同一個，但這不是必須的。比如，我們可以創建另一個字典 s:Math，它保存了一個 PI 鍵，爲示區別，這個 PI 保存的圓周率精度大一些：

```
" >>File: ~/.vim/vimllearn/funcref.vim

let s:Math = {}
let s:Math.PI = 3.14159
let s:Math.Area = s:dOperator.area
echo s:Math.Area(2)
```

請觀察 s:dOperator.area(2) 與 s:Math.Area(2) 計算結果的不同，表明後者調用時 self.PI 確實用到了 s:Math.PI 的值，而不是 s:dOperator.PI 的值。而且，在 s:Math 中的函數名 Area 不一定要與最初定義時所用的 area 相同。但是函數體內用到的 PI 鍵名，必須相同。

如果把 s:dOperator.area 這個函數（引用）賦值給普通變量（非字典鍵），會發生什麼情況呢？嘗試在腳本末尾繼續添加如下代碼並加載運行：

```
let g:Fnr = s:dOperator.area
echo g:Fnr(2)
```

結果它會報 E725 錯誤，提出不能在沒有字典的情況下調用具有 dict 屬性的函數。這似乎很好理解，因爲在 area() 函數體內，用到了 self.PI，沒有字典的話，這個 self 就無所引用了。實際上，即使在函數體內沒有到用 self ，也不能繞過字典去調用字典函數。比如原來的 s:dOperator.sum() 就沒用到 self，但如下代碼也時非法的：

```
let g:Fnr = s:dOperator.sum
```

```
echo g:Fnr(1,2,3,4)
```

在爲 g:Fnr 賦值時不會出錯，在調用 g:Fnr() 時纔出錯。所以 vim 是通過 dict 這個函數屬性來檢測調用合法性的，因爲這種函數體內有可能用到 self，提前終止潛在的錯誤，總是更安全的設計。而且，既然用到 dict ，就意味着大概率會用到 self，否則將一個非 dict 屬性的函數保存在字典中，是很無趣的（雖然合法）。以下語句卻不會出錯：

```
let g:Fnr = s:dOperator.sumFnr
echo g:Fnr(1,2,3,4)
```

因爲 s:dOperator.sumFnr 所引用的函數其實是 s:sum()，它在定義時未指定 dict 屬性。所以 s:dOperator.sunFnr 只起到一個傳遞變量值的中介作用，g:Fnr 也是對 s:sum() 的函數引用，當然也就可以直接調用了。

普通函數的字典屬性

上面在定義 s:dOperator.sum 與 s:dOperator.area （對匿名函數的引用）時，並未顯式寫出 dict 屬性。這只是‘:function’定義字典函數時的語法糖，vim 會自動添加 dict 屬性。

定義普通函數時也可以指定 dict 屬性，例如我們另外寫個計算矩形面積的函數：

```
1 function! s:area(width, height) dict
2     return a:width * a:height
3 endfunction
4
5 " echo s:area(3, 4) |" 出錯
6
7 let s:Rect = {}
8 let s:Rect.area = function('s:area')
9 echo s:Rect.area(3, 4) |" 正確
```

但是，由於 s:area() 函數是 dict 屬性的，所以直接調用 s:area() 會出誤。必須把它（的引用）放在一個字典中，如上爲此專門建了個空字典變量 s:Rect，將函數引用保存在其 area 鍵名中，才能調用 s:Rect.area()。

因此，當一個普通函數用了 dict 屬性，卻沒用到 self 特性，好像用處不是很大，反而限制了其正常使用。爲此，將 s:area() 函數重新定義如下：

```
1 function! s:area() dict
2     return self.width * self.height
3 endfunction
4
5 let s:Rect.width = 3
```

```
let s:Rect.height = 4
echo s:Rect.area()
```

取消 s:area() 的函數參數，而將 width 與 height 參數保存在 s:Rect 字典中，然後就可以無參調用 s:Rect.area() 了。這樣，長、寬就相當於矩形（s:Rect）的屬性，而求面積的 area() 就相當於它的方法。這就初具面向對象的特徵了（這將在後續章節中再詳細討論）。

注意這裏的 s:area() 函數體內用到了 self，則在函數頭一定要指定 dict 屬性。反之則不強制要求。

具有 dict 屬性的函數，除了對用字典鍵引用來調用外，也可以用 call() 函數間接調用。之前已經介紹過 call() 函數，其實它還可接收第三個可選參數，按 ':help call()' 介紹其用法是 call(func, arglist [, dict])。如果第一個參數（函數名或函數引用）所指代的函數具有 dict 屬性，第三個參數就應該提供一個字典傳遞給這個函數體實現中的 self 變量。

因此，第二個版本（無參數）的 s:area() 可以這麼調用：

```
echo call('s:area', [], s:Rect)
echo call(function('s:area'), [], s:Rect)
```

這兩條語句都合法，不過由於使用了 s:area 字符串，必須在腳本中才能運行。當 call() 在調用 s:area() 時，s:area() 函數內的 self 也就是 s:Rect 了。

至於第一個版本帶兩個參數的 s:area() 則可以這麼調用：

```
echo call('s:area', [5, 6], {})
echo call('s:area', [5, 6]) |" 出錯
```

將參數收集在一個列表變量中，作爲第二參數傳入。由於函數體內未用到 self ，在第三參數隨便提供一個字典變量就行，即使是個空字典 {}。但若不提供這個字典參數，則會發生運行時錯誤。

直接定義字典函數與間接定義的比較

綜上再小結一下，定義字典函數（引用）有兩種方式。一是直接用一條語句搞定，字典鍵引用了一個匿名函數；二是先定義函數，再將該有名函數的引用賦值給字典鍵。不妨分別稱之爲直接定義與間接定義。

- 直接定義：function dict.method()
- 間接定義：function Method() 與 let dict.method = function('Method')

顯然，直接定義的語法更簡潔方便，請儘量使用這種語法。那麼間接定義的寫法還有沒有什麼存在的意義呢?

首先，這可能是歷史原因。VimL 也是隨 Vim 逐步發展完善起來的，很有可能函數引用的概念先於 dict 屬性與 self 變量的引入。因而也就先有分步寫的字典函數引用，然後纔有一步到位的語法糖寫法。

其次，間接定義的函數引用有更靈活的控制權。直接定義的字典函數必定是匿名函數的引用，且隱含具有 dict 的屬性，不論是否顯式寫出該關鍵詞。這也就意味着不能將直接定義的字典函數引用賦值給普通函數引用變量，那是不能工作的。但在間接定義字典函數時有更多的選擇，在定義函數時可根據需要是否指定 dict 屬性。沒有 dict 屬性的函數引用可以賦值給普通變量。因此，從編碼實踐上建議：

- 直接定義的字典函數，也始終顯式加上 dict 關鍵詞，不要太依賴語言的隱式作用。
- 普通函數，如果實現體中需要用到 self 才加 dict 屬性關鍵詞。

最後，字典鍵名引用有名或匿名函數，會影響調試與錯誤信息。通過示例詳細說明，將以下代碼片斷添加到本節的演示腳本末尾，並用‘:source’重新加載。

```
1  " >>File: ~/.vim/vimllearn/funcref.vim
2
3  function! s:Rect.debug1() dict abort
4      echo expand('<sfile>')
5      Hello Vim, 我在這裏就是個錯誤
6  endfunction
7
8  function! s:debug2() abort
9      echo expand('<sfile>')
10     Hello Vim, 我來這裏也是個錯誤
11 endfunction
12 let s:Rect.debug2 = function('s:debug2')
13
14 function! s:Rect.test() dict " abort
15     echo expand('<sfile>')
16     call self.debug1()
17     call self.debug2()
18 endfunction
19
20 function! s:test() abort
21     echo expand('<sfile>')
22     call s:Rect.test()
23 endfunction
24
25 function! Test() abort
26     echo expand('<sfile>')
27     call s:test()
28 endfunction
29
```

```
30 echo expand('<sfile>')
```

複用原來的字典 s:Rect，增加了兩個函數引用鍵，其中 debug1 是直接定義的，debug2 是間接引用 s:debug2() 的。這兩個函數內隨意加了一行錯誤語句。這在加載腳本時並不會出錯誤，只有實際調用了相應函數纔有機會出錯。然後再定義了一個統一的 s:Rect.test() 函數，在其內調用這兩個 debug 函數。最後還定義了 s:test() 與 Test() 函數。只有 Test() 是全局的，可以在命令行中執行‘:call Test()’查看結果。在執行前先人工分析下這將發生的函數調用鏈：

```
1 全局函數 Test()-->脚本函數 s:text()-->字典函數 s:Rect.test()
2 [1] --> 字典函數 s:Rect.debug1() | 引用匿名函數
3 [2] --> 字典函數 s:Rect.debug2() | 引用 s:debug2() 函數
```

我這裏執行‘:call Test()’後輸出如下，脚本編號與函數編號肯定是依環境不同的：

```
1 function Test
2 function Test[2]..<SNR>77_test
3 function Test[2]..<SNR>77_test[2]..181
4 function Test[2]..<SNR>77_test[2]..181[2]..180
5 Error detected while processing function Test[2]..
6     <SNR>77_test[2]..181[2]..180:
7 line    2:
8 E492: Not an editor command:   Hello Vim, 我在這裏就是個錯誤
9 function Test[2]..<SNR>77_test[2]..181[3]..<SNR>77_debug2
10 Error detected while processing function Test[2]..
11     <SNR>77_test[2]..181[3]..<SNR>77_debug2:
12 line    2:
13 E492: Not an editor command:   Hello Vim, 我來這裏也是個錯誤
```

其中，常規字體是各函數內 echo expand('<sfile>') 的正常輸出，紅字部分是錯誤語句觸發的輸出，即 vim 自動給出的錯誤提示信息。主要是觸發 E492 這個錯誤，它說 Helle Vim 不是編輯器的有效命令。並在之前先打印出錯時所在的函數名與行號。重點關注一下函數名的表示方法，例如在 s:Rect.debug1() 出錯時的位置信息：

```
  function Test[2]..<SNR>77_test[2]..181[2]..180:
```

對比之前的分析，第一層調用是全局函數 Test，中括號 [2] 表示在第二行調用下一層函數，即 s:test()，它被轉化成 <SNR>77_test 函數名，然後第二行再調用 s:Rect.test()，這是匿名函數，所以只能打印出編號 181，然後繼續調用 s:Rect.debug1()，它也是匿名函數，也只打印出編號 180。到這個函數就出錯了，沒能再調用其他函數，出錯行號另起一行打印出來。

在 s:Rect.debug2() 出錯時的位置信息類似：

```
function Test[2]..<SNR>77_test[2]..181[3]..<SNR>77_debug2:
```

只不過在倒數第二層的行號從第二行改爲了第三行，最後一個函數名打印出了實際所引用的函數名 <SNR>77_debug2 ，也就是腳本中的 s:debug()。

這有什麼差別呢? 試想我們若用 VimL 開發實用功能（主要是插件時），調用鏈經常也會這麼長或者更長。當 vim 報錯時，給出一長串錯誤提示，我們第一反應是想知道哪裏出錯了，最終出錯在哪個函數中。這反映在出錯信息的最後一個調用函數，但是像 s:Rect.debug1() 這樣的直接定義的字典函數，vim 只打印個 180 編號，可能完全不知所云。而像 s:Rect.debug2() 這個間接定義的字典函數，它會打印出函數名。即使你也不知腳本編號，那也是有🄵可循，比如用‘:scriptnames’檢查。而且在實踐中，你也不可能在很多不同腳本中都定義了相當的函數，那麼不用檢查腳本編號也基本能定位錯誤了。

還有重要一點，在開發 VimL 腳本過程中，如果修改 Bug 後重新加載腳本，那直接定義的字典函數所引用的匿名函數編號是會變化的。因爲它相當於重新定義了另一個匿名函數併爲字典鍵賦值，而原來那個匿名函數再無引用無可訪問就會自動釋放（垃圾回收機制）。但是，腳本編號並不會改變，除非大重構把文件名也改了。這種編號的變化性對查 Bug 也多少會有影響的。

順便提一下，也許你也注意到了，vim 自動打印的出錯位置信息，其實就是 <sfile> 的值。如果用在函數中，那就是運行到該處時完整的調用鏈字符串；在不同時刻從不同入口調用時還可能給出不同的值。但如果用在函數外，那就只能是在腳本文件中，<sfile> 就表示腳本文件名（故不能直接用在命令行中）。這也是 sfile 這個單詞意義的來源。不過你也可以將腳本整體理解爲一個函數（也是一個執行單元），其“函數名”顯然就是腳本名了。

還有一點得注意，在定義 s:Rect.test() 函數時，沒有加 abort 屬性。按之前的建議，定義函數時始終加 abort 是良好的習慣，因爲它會在出錯時立即終止運行，避免更多的錯亂。不過在這裏，如果有 abort 屬性，它在調用 self.debug1() 出錯後就立即終止，self.debug2() 也就沒機會調用了。由於我們想對比出錯信息，要求觸發所有錯誤，因而特意取消 abort 屬性。

5.4 * 閉包函數

自 Vim8，進一步擴展與完善了函數引用的概念，並增加了對閉包與 lambda 表達式的支持。請用‘:version’命令確認編譯版本有 +lambda 特性支持。

閉包函數定義

學習 Vim 新功能，在線幫助文檔是最佳資料。查閱 Vim8 的‘:help :function’，可發現在定義函數時，除了原有的幾個屬性 range，abort，dict 外，還多了一個 closure 屬性。這就是定義閉包函數的關鍵字。並給出了一個示例，我們先將其複製到一個腳本中並執行:

```
1 " >File: ~/.vim/vimllearn/closure.vim
```

```
2
3  function! Foo()
4      let x = 0
5      function! Bar() closure
6          let x += 1
7          return x
8      endfunction
9      return funcref('Bar')
10 endfunction
```

這裏有幾點需要說明:

- 函數可以嵌套了，在一個函數體內可以再定義另一個函數。
- 內層函數 Bar() 指定了 closure 屬性，就是將其定義爲閉包函數。
- 在內層閉包函數 Bar() 中，可以使用外層環境函數 Foo() 的局部變量 x。
- 外層函數返回的是內層函數的引用。
- 當 Foo() 函數返回後，在 Bar() 內仍然可正常使用局部變量 x。

現在來使用這個閉包，可在命令行中直接輸入以下語句試運行:

```
let Fn = Foo()
echo Fn()
echo Fn()
echo Fn()
```

可見，在每次調用 Fn()，也就是調用 Bar() 時，它會返回遞增的自然數，在兩次調用之間，會記住變量 x 的值。對比普通函數，當其返回後，其部分變量就離開作用域不再可見，每次調用必須重新創建與初始化局部變量。而 Bar() 函數能記住 x 變量的狀態，就是由於 closure 關鍵字的作用。

除些之外，Bar() 就與普通函數一樣了。特別地，它的函數全名就是 'Bar'，即它也是個全局函數，也可以直接在命令行調用。如下語句依然正常地輸出遞增自然數:

```
echo Bar()
echo Bar()
echo Fn()
```

另外必須指出的是，在 Foo() 函數內創建 Bar() 引用時，用的是 funcref() 函數，而不是 function() 函數。funcref() 也是 Vim8 才引入的內置函數，它與之前的 function() 函數功能一樣，也就是創建一個函數引用。只有一個差別，function() 只簡單地按函數名尋找它所 "引用" 的函數，而 funcref() 是按真正的函數引用尋找目標函數。這其中的差別只在原函數被重定義了才能體現。

例如，我們再用 function() 創建一個類似的閉包函數引用，爲示區別每次遞增 2。將以下代碼附加在原腳本之後，再次加載運行。

```
" >>File: ~/.vim/vimllearn/closure.vim

function! Goo()
    let x = 0
    function! Bar() closure
        let x += 2
        return x
    endfunction
    return function('Bar')
endfunction

let Gn = Goo()
echo Gn()
echo Gn()
echo Bar()
echo Gn()
```

初步看來，Goo() 函數能與 Foo() 完全一樣地使用，獲取一個閉包引用，依次調用，並且可與所引函數 Bar() 交替調用，也能保持正確的狀態。

但要注意，在 Goo() 函數內定義的閉包函數也是 Bar() 。所以在每次調用 Goo() 或 Foo() 都會重新定義全局函數 Bar()。如果用 function() 獲取 Bar() 的引用，它就是使用最新的函數定義。如果用 funcref() 獲取 Bar() 的引用，它就一直使用當時的函數定義。

例如，我們直接在外面再次重定義一下 Bar() 函數：

```
function! Bar()
    return 'Bar() redefined'
endfunction

echo Bar()
echo Fn()
echo Gn()
```

運行結果表明，Fn() 能繼續遞增數值，但 Gn() 卻調用了重新定義的函數，失去了遞增的原意。

所以，爲了保證閉包函數的穩定性，務必使用新函數 funcref() ，而不要用舊函數 function()。當然，function() 函數除了爲保證兼容性外，應該也還有其適合場景。

另外，非常不建議直接調用閉包函數，應該堅持只通過函數引用變量來調用閉包。但是，目前的 VimL 語法，似乎沒法完全阻止直接調用閉包。因爲 ‘:function’ 定義的是函數，而非變量，不能爲函數名添加 ‘l:’ 前綴來限制其作用域。可以加 ‘s:’ 定義爲腳本範圍

的函數，但它仍然可以從外部調用（相對於創建閉包的 Foo() 環境而言）。一個建議是爲閉合函數名添加一些特殊後綴，給直接書寫調用增加一些麻煩。

閉包變量理解

閉包函數的關鍵是閉包變量，也就是閉包函數內所用到的外部局部變量。

其實，在一個函數內使用外部變量是很平凡的。比如：

```
let s:x = 0
function! s:Bar() " closure
    let s:x += 1
    return s:x
endfunction
```

這裏只用以前的函數知識定義了一個 s:Bar() 腳本函數，它用到腳本局部變量 s:x。每次調用 s:Bar() 時，也能遞增這個變量。似乎也能達到之前閉包函數的作用，然而這只是幻覺。因爲 s:x 不是專屬於 s:Bar() 函數的，即使也限制了腳本作用域，也能被腳本中其他函數或語句修改。

而之前閉包函數 Bar() 的變量 x ，原是 Foo() 函數內創建的局部變量。當 Foo() 函數返回後，這個局部變量理論上要釋放的，也就無從其他地方再次訪問，只能通過 Bar() 這個即時定義的閉包函數才能訪問。

所以，閉包變量既是外部變量，更重要的是外部的局部變量。這才能保證閉包函數對於閉包變量的專屬訪問。也因爲這個原由，在頂層（腳本或命令）定義的函數不能指定閉包屬性。如上定義 s:Bar() 函數時若加上 closure 將會直接失敗。而一般只能嵌套在另一個函數中定義閉包函數，這個外層函數有的也叫工廠函數。工廠函數爲閉包提供一個臨時的局部環境，閉包變量先是在工廠函數中創建並初始化，而在閉包函數裏面則是自動檢測的，凡用到的外部局部變量都會轉爲閉包函數。當然了，在工廠函數或閉包函數內都可以有其他各自的普通局部變量。

在工廠函數內創建閉包函數時，閉包變量就成爲了閉包函數的一個內部屬性。每次調用工廠函數時，會創建閉包函數的不同副本，也就會有相應閉包變量的不同副本。也就是說，每次創建的閉包函數會維護各自的狀態，互不影響。

爲說明這個問明，再舉個例子。比如把上面實現的遞增 1 與遞增 2 的兩個閉包放在一個工廠函數內創建，借用列表同時返回兩個閉包：

```
function! FGoo(base)
    let x = a:base
    function! Bar1_cf() closure
        let x += 1
        return x
    endfunction
```

```
    function! Bar2_cf() closure
        let x += 2
        return x
    endfunction
    return [funcref('Bar1_cf'), funcref('Bar2_cf')]
endfunction

echo 'FGoo(base)'
let [Fn, X_] = FGoo(10)
echo Fn()
echo Fn()
echo Fn()
let [X_, Gn] = FGoo(20)
echo Gn()
echo Gn()
echo Gn()
echo Fn()
echo Fn()
```

另一個改動是給工廠函數傳個參數，讓其成爲閉包遞增的初值。在調用工廠函數時，也利用列表解包的語法，同時獲得返回的兩個閉包函數（引用）。第一次 let [Fn, X_] = FGoo(10) 用 10 作爲初值，且只關心第一個閉包 Fn ，第二個 X_ 只作爲佔位變量棄而不用。在執行 Fn() 數據後，第二次調用 let [X_, Gn] = FGoo(20) 傳入另一個初值，且只取第二個閉包 Gn。然後可以發現這兩個閉包能並行不悖地執行。這說明閉包變量 x 雖然是在 FGoo 中創建，卻不隨之保存，而是保存在各個被創建的閉包函數中。

偏包引用

自 Vim8 ，不僅爲創建函數引用增加了一個全新的內置函數，而且還爲 function() 與 funcref() 升級了功能。除了提供函數名外，還可以提供一個可選的列表參數，作爲所引用函數的部分的參數。如此創建的函數引用叫做 partial ，這裏將之稱爲偏包。

請看以下示例：

```
function! Full(x, y, z)
    echo 'Full called:' a:x a:y a:z
endfunction

let Part = function('Full', [3, 4])
call Part(5)
```

首先定義了一個 “全” 函數 Full()，它接收三個參數，不妨把它認爲是三維空間上的座標點。假設有種需求，平面座標已經是固定的了，只是還要經常改變高座標。這時就可用 function() （或 funcref()）創建一個偏包，將代表固定平面座標的前兩個參數放在一個列表變量中，傳給 function() 的兩個參數。然後調用偏包時，就不必再提供那已固定的參數，只要傳入剩余參數即可。如上調用 Part(5) 就相當於調用 Full(3, 4,5) 。

function() 的第一參數，不僅可以是函數名，也可以是其他函數引用。於是偏包的定義可以鏈式傳遞（有的叫嵌套）。例如：

```
let Part1 = function('Full', [3])
let Part2 = function(Part1, [4])
call Part2(5) |" => call Full(3, 4, 5)
```

須要注意的是，在創建偏包時，即使只要固定一個參數，也必須寫在 [] 中，作爲只有一個元素的列表傳入。

爲什麼這叫偏包，因爲偏包本質上是個自動創建的閉包。例如以上爲 Full() 創建的偏包，相當於如下閉包：

```
function! FullPartial()
    let x = 3
    let y = 4
    function! Part_cf(z) closure
        let z = a:z
        return Full(x, y, z)
    endfunction
    return funcref('Part_cf')
endfunction

let Part = FullPartial()
call Part(5)
```

至於用 function() 創建通用偏包的功能，可用如下閉包模擬：

```
function! FuncPartial(fun, arg)
    let l:arg_closure = a:arg
    function! Part_cf(...) closure
        let l:arg_passing = a:000
        let l:arg_all = l:arg_closure + l:arg_passing
        return call(a:fun, l:arg_all)
    endfunction
    return funcref('Part_cf')
endfunction
```

```
10
11 let Part = FuncPartial('Full', [3, 4])
12 call Part(5)
```

以上的語句 let l:arg_all = l:arg_closure + a:000 表明了在調用偏包時，傳入的參數是串接在原來保存在閉包中的參數表列之後的。其實，那三條 let 語句創建的中間變量是可以取消的，只須用 return call(a:fun, a:arg + a:000) 即可。其中 a:fun 與 a:arg 變量來源於外部工廠函數 FuncPartial() 的參數，將成爲閉包變量，而 a:000 則是在調用閉包函數時傳入的參數。

這個 FuncPartial() 只爲說明偏包與閉包之間的關係，請勿實際使用。另請注意這兩概念的差別，閉包是函數，偏包是引用，偏包是對某個自動創建的閉包的引用。

創建函數引用尤其是偏包引用的 function() 與 funcref() 函數，不僅可以接收額外的列表參數，還可接收額外的字典參數。這與 call() 函數的參數意義是一樣的。當需要創建引用的函數有 dict 屬性時，傳給 function() 的字典參數就將傳給目標函數的 self ，實際上也將該字典升格爲閉包變量。之後再調用所創建的偏包引用時，就不必再指定用哪個字典當作 self 了。

不過 function() 與 call() 的參數用法也有兩個不同：

• call() 至少要兩個參數，即使目標函數不用參數，也要傳 []。function() 默認只要一個參數即可。

• function() 可以直接傳字典變量當作第二參數，不必限定第二參數必須用列表，不必用 [] 空列表作佔位參數。當然也可以同時傳入列表與字典參數，此時應按習慣不要改變參數位置。

lambda 表達式

lambda 表達式用於創建簡短的匿名函數，其語法結構如：let Fnr = args -> expr。幾個要點：

• 整個 lambda 表達式放在一對大括號 {} 中，其間用箭頭 -> 分成兩部分。

• 箭頭之前的部分是參數，類似函數參數列表，多個參數由逗號分隔，也可以無參數。無參數時箭頭也不可以缺省，如 {-> expr} 形式。

• 箭頭之後是一個表達式。該表達式的值就是以後調用該 lambda 時的結果。這有點像函數體，但函數體是由多個 ex 命令語句構成。lambda 的 “函數體” 只能是一個表達式。

• expr 部分在使用 args 的參數時，不要加 ‘a:’ 參數作用域前綴。

• 在 expr 部分中還可以使用整個 lambda 表達所處作用域內的其他變量，如此則相當於創建了一個閉包。

• 一般需要將 lambda 表達式賦值給一個函數引用變量，如此才能通過該引用調用 lambda 。也就是說 lambda 表達式自身的值類型是 v:t_func。

舉個例子，假設有如下定義的函數：

```
1 function! Distance(point) abort
```

```
2       let x = a:point[0]
3       let y = a:point[1]
4       return x*x + y*y
5   endfunction
```

這裏假設用只含兩個元素的列表來表示座標上的點，該函數的功能是計算座標點的平方和，這可作爲距離原點的度量。幾何上的距離定義其實是平方和再開根號，不過開根號的浮點運算效率低，尤其是相對整數座標來說。所以在滿足程序邏輯的情況下，可以先不開這個根號，比如只在最後需要顯示在 UI 上纔開這個根號。

然而無關背景，這個函數或許很重要，但實現很簡單，實際上也可用 lambda 來代替：

```
let Distance = {pt -> pt[0] * pt[0] + pt[1] * pt[1]}
```

當然了，這兩段代碼不能同時存在，因爲函數引用的變量名，不能與函數名重名。分別執行這兩段，測試‘:echo Distance([3,4])’能輸出 25 。

前面說過，閉包函數不能在腳本（或命令行）頂層定義，但 lambda 表達式可以。因爲 lambda 表達式其實是相當於創建閉包的外層工廠函數（及其調用），那當然是可以寫在頂層了。不過就這個 Distance 實例，並未用到外部變量，可不必糾結是否閉包。

然後，我們利用這個函數寫一個具體功能，比如計算一個三角形的最大邊長。輸入參數是三個點座標，輸出最大邊長（的平方）：

```
1   function! MaxDistance(A, B, C) abort
2       let [A, B, C] = [a:A, a:B, a:C]
3       let e1 = [A[0] - B[0], A[1] - B[1]]
4       let e2 = [A[0] - C[0], A[1] - C[1]]
5       let e3 = [B[0] - C[0], B[1] - C[1]]
6       let d1 = Distance(e1)
7       let d2 = Distance(e2)
8       let d3 = Distance(e3)
9       if d1 >= d2 && d1 >= d3
10          return d1
11      elseif d2 >= d1 && d2 >= d3
12          return d2
13      else
14          return d3
15      endif
16  endfunction
```

這裏，直接用單字母表示參數了，似乎有違程序變量名的取名規則。不過這也要看具體場景，因爲這是解決數學問題的，直接用數學上習慣的符號取名，其實也是簡潔又不失

可讀性的。該函數先從頂點座標計算邊向量，再對邊向量調用 Distance() 計算距離，返回其中的最大值。

如果 Distance 是上面定義的函數版本，這個 MaxDistance() 直接可用。比如在命令行中試行：‘:echo MaxDistance([2,8], [4,4], [5,10])’ 將輸出 37 。

但如果是用 lambda 表達式版本，將 let Distance = ... 寫在全局作用域中，那麼在調用 MaxDistance() 時再調用 Distance() 就會失敗，指出函數未定義的錯誤。把這個 lambda 表達式寫在 MaxDistance() 開頭，剩余代碼才能正常工作。

不過這個困惑與 lambda 無關，只是作用域規則。解析 let d1=Distance(e1) 時，如果 Distance 不是一個函數名，就會嘗試函數引用。然而在函數內的變量，缺省前綴是 ‘l:’ ，所以它找不到在外部定義的 g:Distance。基於這個原因，個人非常建議在函數內部也習慣爲局部變量加上 ‘l:’ 前綴，這樣就能使函數引用變量名與函數名從文本上很好地區分，避免迷惑性出錯。

同時，這也說明了 lambda 的習慣用法，一般是在需要用的時候臨時定義，而不是像常規函數那樣預先定義。

最後提一下，lambda 作爲匿名函數，vim 對其表示法是 <lambda>123 ，與上一章介紹的字典匿名函數一樣，只是在編號前再加 <lambda> 前綴，同時這兩套編號相互獨立。

小結

偏包與 lambda 表達式，本質上都是閉包，而閉包也一般只以其函數引用的形式使用。Vim8 引入這些編程概念的一個原因，是爲了方便在局部環境中創建回調函數，與異步、定時器等特性良好協作。

5.5 自動函數

自動加載函數（:h autoload-functions）自 Vim7 版本就支持了。不過它涉及的機制就不僅僅是函數本身了，所以放在本章之末再討論。其實自動加載機制已經在第一章就作爲 VimL 語言的一個特點介紹過了，請回頭複習一下，在那裏已經將自動函數的加載流程描敘的比較細緻了。

本節繼續講解有關自動函數的定義與使用。

函數未定義事件

自動加載函數的作用是，當安裝的插件比較多時，不應該在啓動 Vim 時全部加載（通過 vimrc :source 調用或放在 plugin/ 目錄下），而應只在需要用到時才加載。當然，自定義命令與映射，相當於面向用戶操作的 UI，那是應該在一開始就加載好（保證有定義）。但是複雜功能的命令與映射，往往是調用函數完成實際功能的，然後所調用的函數又可能只是個入口函數，其中又會涉及一堆相關功能的函數。那麼這些函數的定義就可以延後加載，只在首次用到時觸發加載，就能達到優化 Vim 啓動速度的命令。

在 Vim7 版本之前，用戶可以利用 FuncUndefined 這個自動事件來實現延後加載腳本的目的。例如，假設在任一 &rtp 目錄下的 plug/ 中有如下腳本：

```
" >File: ~/.vim/plugin/delaytwice.vim

if !exists('s:load_first')
    command -nargs=* MYcmd  call DT_foo(<f-args>)
    nnoremap <F12> :call DT_foo()<CR>
    execute 'autocmd FuncUndefined DT_* source '
            . expand('<sfile>')
    let s:load_first = 1
    finish
endif
if exists('s:load_second')
    finish
endif

function! DT_foo() abort
    " TODO:
endfunction
function! DT_bar() abort
    " TODO:
endfunction

let s:load_second = 1
```

這個腳本將分兩步加載。首先，由於它位於 plugin/ 子目錄，故在 Vim 啓動時就會讀取。在這第一次加載時，會進入 if !exists(‘s:load_first’) 分支，該分支應該很短，只定義了命令與映射，並用一個 ‘s:’ 變量標記已加載過一次後直接結束。關鍵是定義了自動事件 FuncUndefined，此後當調用了未定義函數且該函數名匹配 DT_* ，就會重新加載這個腳本。第二次加載時，會跳過 if !exists(‘s:load_first’) 分支，繼續加載後續代碼，完成相應函數的定義。

後面那個 if exists(‘s:load_second’) 分支，是爲了避免第三次或更多次的加載。對於其他普通腳本，也可用這個機制防止重複加載，不過僅爲實現延時加載，這個分支是不必要的。一般地，如果腳本中主要是用 ‘:function!’ 命令定義一些函數，重複加載也沒有太大壞處，畢竟重複定義而覆蓋的函數與原來的是一樣。但是若腳本中需要維護某些 ‘s:’ 局部變量（尤其是較複雜的字典對象）的狀態，重複加載腳本就會導致這些變量的重新初始化，可能就不是想要的，這就需要避免加載。這與延時加載是兩個理念，延時加載是有意地設計爲加載第二次。

實際上，這延時加載的兩部分，可以分別寫在不同的兩個腳本中。這對於非常大的腳本，可能還能進一步提高 Vim 啓動速度。因爲按上例，寫在一個腳本中，雖然 vim 不必解釋後半部分的代碼，但畢竟首先還是要打開整個腳本文件的。因此，將第一部分定義的命令、映射與自動事件放在 plugin/ 目錄下，令其在 Vim 啓動時就加載。第二部分的函數定義（主體內容，長）放在另一個目錄下，只要不會被 vim 在啓動階段讀取就可，例如不妨就放在 autoload/ 子目錄下。拆分結果示例如下：

```
" >File: ~/.vim/plugin/delaytwice.vim
command -nargs=* MYcmd  call DT_foo(<f-args>)
nnoremap <F12> :call DT_foo()<CR>
execute 'autocmd FuncUndefined DT_* source '
        / . expand('~/.vim/autoload/delaytwice.vim')
```

```
" >File: ~/.vim/autoload/delaytwice.vim
function! DT_foo() abort
    " TODO:
endfunction
function! DT_bar() abort
    " TODO:
endfunction
```

不過在拆分時，有一行代碼要注意作相應修改。就是在定義 FuncUndefined 事件時，需要加載正確的（另一個）文件路徑。上例是硬編碼寫入了對應的全路徑。而在前面的單文件的版本中，可用 <sfile> 表示本腳本文件名。當然，在後面這個拆分版本中，若按某種規範存在相對路徑中，也是可以避名硬編碼的，如用以下語句代替：

```
  expand('<sfile>:p:h') . '/../autoload/' . expand('<sfile>:p:t')
```

此外還須說明的是，用這種（手動）延時加載方案時，所定義的函數名最好用統一的前綴（或後綴），方便在定義 FuncUndefined 事件時指定相應的模式匹配，儘量使該匹配不擴大影響，也能保證所用函數能正確延時加載到。

自 Vim7 版本後，有了自動延時加載機制，就不必用戶自己實現手動延時加載方案了。不過以上的手動延時方案，有助於理解 Vim 的自動加載機制。另外單文件版本的示例可能仍有意義，不那麼複雜的腳本若不想拆分多個文件，就可按此例用 FuncUndefined 事件實現。

自動加載函數的定義

自動加載機制，與上節討論的拆分版的延時加載方案示例類似，不過有以下幾點不同：

- 不必再寫 FuncUndefined 事件；
- 將函數名前綴的 DT_ 改爲 DT#

- 將定義函數的那個腳本文件名也改爲 autoload/DT.vim

```
" >File: ~/.vim/plugin/delaytwice.vim
command -nargs=* MYcmd  call DT#foo(<f-args>)
nnoremap <F12> :call DT#foo()<CR>
```

```
" >File: ~/.vim/autoload/DT.vim
function! DT#foo() abort
    " TODO:
endfunction
function! DT#bar() abort
    " TODO:
endfunction
```

這樣就可以了，不必再關注 DT#foo() 函數有沒有定義，什麼時刻定義，在任何地方直接使用就可以了。vim 能自動識別函數名中間包含 # 符號的函數，當作自動加載函數處理，將 # 符號之前的部分視爲腳本文件名，在函數未定義時，自動到 &rtp 的 autoload/ 子目錄下查找。

所以關鍵是要讓函數名的 # 前綴與文件名保持一致，也可以不改 delaytwice.vim 的文件名，而將函數名改爲 delaytwice#foo()。這種函數名的首字符允許是小寫，畢竟全局函數名首字母大寫的規則，主要是爲了避免與內置函數衝突。

自動加載函數名中可以有多個 # 符號分隔，對應於 autoload/ 子目錄下各級路徑：

```
{&rtp}/autoload/sub1/sub2/filename.vim
function sub1#sub2#filename#func_name()
```

當 vim 加載含有 # 函數定義的腳本文件時，如果發現函數名前綴與文件路徑不相符，就會報錯，即無法順利完成該函數的定義。不過事實上它只檢查是否在 autoload/ 目錄下的相對路徑，至於 autoload/ 之上的父目錄是否在 &rtp 中並不強制檢測。因爲在正式應用環境下，該腳本文件已經是從 &rtp 中搜索到的。而另一方面，在開發測試時，你只要建立相應的目錄層次，把文件扔到（某個工程）autoload/ 子目錄下，即使暫沒把工程目錄加到 &rtp 下，在編輯這個腳本時，也可以用 ‘:source %’ 加載當前文件進行測試，並不會因爲它還不在 &rtp 中就失敗。

當有了 # 函數的自動加載機制，那是否可以與 FuncUndefined 事件聯用協作呢？一般情況下沒有必要。但假設一種情況，如果按目前流行的方式用插件管理插件從 github 安裝插件的話，一般是將每個插件放在獨立的目錄中，每個插件目錄都加入了 &rtp 中。這樣如果你真的很狂熱地安裝了許多插件，你的 &rtp 路徑列表將變得很長。&rtp 路徑在 vim 運行是至關重要，不僅這裏介紹的自動加載函數，其他許多功能都要從 &rtp 中查找。如果某個插件的主要功能只是提供了 autoload/ 腳本，或許就可以嘗試合併 &rtp，自己再寫一個 FuncUndefined 事件，從其他地方加載腳本。

那麼就要注意 # 函數內置的自動加載時機，與 FuncUndefined 事件的觸發，先後關係如何，避免一些可能的衝突。下面做一個試驗來探討之。

首先，在 ~/.vim/autoload/delaytwice.vim 腳本末尾加入如下一些輸出語句，用以跟蹤該腳本被加載的情況：

```
function! delaytwice#foo() abort
    echo 'in delaytwice#foo()'
endfunction

" bar:
function! delaytwice#bar() abort
    echo 'in delaytwice#bar()'
endfunction
echo 'autoload/delaytwice.vim loaded'
```

然後，再自定義一個 FuncUndefined 事件：

```
execute 'autocmd FuncUndefined *#*  call MyAutoFunc()'

function! MyAutoFunc() abort
    echo 'in MyAutoFunc()'
    " TODO:
endfunction
```

它也匹配任何中間含 # 符號的函數名，但假設它們（有些）沒放在 &rtp 中，所以需要寫個入口函數從其他地方查找並加載定義文件。將該代碼放在 plugin/ 下某個文件中，便於在每次啓動 vim 時自動執行，保證該事件已定義。

接下來就可以測試了，重啓 vim （或打開 vim 的另一個實例會話），在命令行執行如下命令，其輸出也附於其後：

```
: call delaytwice#foo()
autoload/delaytwice.vim loaded
in delaytwice#foo()
```

這說明只觸發了 vim 內置的自動加載機制，它自動加載了 delaytwice.vim 文件，然後 delaytwice#foo() 函數就是已定義了，就可調用該函數了，不會再觸發 FuncUndefined 事件。

再重啓一個 vim ，在命令行調用一個在該文件中並不存在的函數，比如將 foo() 小寫誤寫成了大寫 Foo()，其輸出如下：

```
: call delaytwice#Foo()
autoload/delaytwice.vim loaded
```

```
in MyAutoFunc()
autoload/delaytwice.vim loaded
E117: Unknown function: delaytwice#Foo
```

從結果可分析出，vim 仍是先按自動加載機制，找到 delaytwice.vim 並加載，然後再嘗試調用 delaytwice#Foo()，它仍是個未定義函數。這二次調用時，才觸發 FuncUndefined 事件。當然我們這裏自定義的 MyAutoFunc() 並沒做實際工作，並不能解決函數未定義問題。於是 vim 再按自動加載機制，找到並加載 delaytwice.vim。加載兩次後仍未解決問題，vim 就報錯了。

當然了，如果在 &rtp 中並沒有找到 delaytwice.vim 或者調用 ':call nofile#foo()'，它只出輸出 in MyAutoFunc() 這行以及錯誤行。但它顯然是遍歷過一次 &rtp 未找到相應文件，才觸發 FuncUndefined 事件的。

現在，又假設不自定義 FuncUndefined 事件與 MyAutoFunc() 處理函數，只按 vim 的自動加載機制，如果調用了在自動加載文件中其實並未定義的函數，會是什麼情況呢：

```
: call delaytwice#Foo()
autoload/delaytwice.vim loaded
autoload/delaytwice.vim loaded
E117: Unknown function: delaytwice#Foo

: call delaytwice#bar()
in delaytwice#bar()

: call delaytwice#Bar()
autoload/delaytwice.vim loaded
E117: Unknown function: delaytwice#Bar
```

可見，第一次調用 delaytwice#Foo() 時，加載了兩次腳本，其內的 delaytwice#foo() 與 delaytwice#bar() 就是已定義的，可正常使用了。然後每次誤用 delaytwice#Foo() 或 delaytwice#Bar() 都會再觸發加載一次腳本。

綜上，可得到如下結論：

• vim 會記錄已加載的腳本文件，當調用自動加載函數時，若分析自動加載函數所對應的自動加載腳本並未加載，就會先搜索並加載相應的腳本，再次調用原函數。

• 在自動加載腳本已加載或未找到相應腳本的情況下，調用未定義的自動加載函數纔會觸發 FuncUndefined 事件，會先調用自定義的事件處理函數，若無法觸發，再次搜索加載相應的腳本。

自動函數與其他函數的比較

首先，要明確一件事，自動加載函數是在全局作用域的。也就是相當於全局函數，可以在任何地方使用。

但是，它又有某些局部函數的作用。比如，可以在兩個自動加載腳本中定義“相同”的函數，onefile#foo() 與 another#foo()。但實際上它們仍是兩個不同的函數，因爲包含 # 在內的整個字符串 onefile#foo 與 another#foo 纔是它們的函數名。它們只是名字上包含相同後綴的相似函數而已。僅管如此，能在不同文件（插件）中，利用相同的詞根定義相同（或相似）功能的不同實現，也是很有意義的，增加代碼可讀性與維護。

在爲自動函數定義函數引用時，也要使用其全名。同時，自動加載函數它是函數，不是變量。所以，在定義 onefile#foo() 的 onefile.vim 文件中，還可以定義 s:foo 變量。但最好不要這樣增加混亂，除非將其定義爲同名函數的引用，如‘:let s:foo = function('onefile#foo')’。因爲即使在同一個文件中，也必須使用包含 # 的函數全名，它可能很長，使用不太方便，所以定義一個局部於‘s:’的函數引用，是有意義的。

除了函數名可以加 # 符號實現自動加載機制，全局變量名也可以加 # 符號。但是隻有當這個變量用於右值，如‘:let default = g:onefile#default’纔會觸發搜索加載相應腳本（onefile.vim）。但用於左值，如‘:let g:onefile#default = 5’卻並不會觸發加載腳本。這個區別其實是爲了讓用戶在觸發加載 onefile.vim 之前，就能設置 g:onefile#default 的值，那可作爲相關插件的用戶配置變量，比之前慣用的 g:onefile_default 變量名似乎更有意義，更像 VimL 風格。

正因爲 # 符號也可用於變量名，才千萬注意不要將一個函數引用變量保存在 # 變量名中（包括 lambda 表達式），雖然那是合法的，但非常不建議如此混亂。只有真的有意設計開放給用戶的重要配置才定義爲 # 全局變量，並始終加上‘g:’前綴。而函數名是不能加‘g:’前綴的，如此容易直觀地區分。

總之，自動加載函數是 VimL 中一個極優秀的設計。如果說函數引用是注重從內部管理，那麼自動加載函數則注重從外部管理。善加利用，可極大增強 VimL 代碼的健壯性與可維護性。若說有什麼缺點的話，那就是自動加載函數名可能太太太長了。並且要由用戶來保證函數名前綴與文件名路徑的一致性，如果腳本文件改名了，或移動了路徑層次，手動修改函數名也是一大問題。非常期望在後續版本中能增加什麼語法糖，能使得在本文件中定義與使用自動加載函數更加簡潔些。

第六章　VimL 內建函數使用

一般實用的語言包括語法與標准庫，畢竟寫程序不能完全從零開始，須站在他人的基石之上。而要開發更有產品價值的程序，更要站在巨人的肩膀上，比如社區提供的第三方庫。

細思起來，VimL 語言的 “標准庫” 包括兩大類：內建命令與內建函數。用戶在此基礎上可自定義命令與自定義函數，再合乎語法地組成起來，以達成所需的功能。第三章簡要地介紹了部分基礎命令，其實那更傾向於 Vim 編輯器的功能。本章要介紹的內建函數，則更傾向於 VimL 語言的功能。

不過本章將會是比較無聊的一章。幫助文檔 ‘:help function-list’ 會按類別列出內置函數，‘:help functions‘ 則會按字母序列出內置函數，可供參考。中文用戶可找一份幫助文檔的中譯本，雖然可能不是最新版本的，不過絕大部分內置函數都應該是穩定向下兼容的。

所以本章不會（也沒必要）羅列所有內建函數，只擇要講些內建函數的使用經驗技法。要查看某個函數的解釋，請直接 ‘:help func_name()’，請注意加一對括號，限定查函數的文檔，否則有可能查到的是同名的命令或選項等。

6.1　操作數據類型

字符串運算

Vim 是文本編輯器，所以處理文本字符串是一重點任務。每個字符在計算機內部都用一個整數表示（即編碼值，與用數字字符組成表示的可讀整數不同概念），具體如何對應取決於編碼系統（有時簡稱編碼）。目前計算機界的趨勢是用 utf-8 編碼表示 Unicode 字符集，因爲它與最早的 ASCII 編碼兼容。一個漢字在此編碼下用 3 個字節表示，英文字符仍用一個字節表示。

- nr2char() 將編碼值轉爲字符
- char2nr() 將字符轉爲編碼值
- str2nr() 將字符串轉爲整數
- str2float() 將字符串轉爲浮點數

由於 VimL 沒有字符串類型，char2nr() 其實是將字符串首字符轉爲編碼值的。默認按 utf-8 編碼獲取編碼值與字符的對應，但可傳入額外參數按其他編碼系統對應。例如：

```
: echo char2nr('中國') |" --> 20013
```

```
: echo nr2char(20013)  |" --> 中
```

整數類型與字符串一般可自動轉換，一般用不上 str2nr() 。但如果從安全考慮習慣主動判斷類型的話，要注意從命令行輸入的參數都是字符串，不是整數。此外，字符串不會自動轉爲浮點數，而是截斷爲整數，所以確實要處理浮點數是，用 str2float() 轉換。

• printf() 格式化字符串

簡單的字符串連接用連接操作符 '.' 即可。要組裝複雜字符串時可用 printf() 函數，通過字符串模式與% 佔位符，插入變量。其用法與 C 語言的 sprintf() 類似，因爲該函數會返回結果字符串，"打印" 字符串用 ':echo' 命令。

• escape() 將字符串中指定的字符用反斜槓 \轉義

• shellescape() 轉義特殊字符以適於 shell 命令

• fnameescape() 轉義特殊字符以適於 Vim 命令，主要用於轉義文件名參數

當組裝字符串用於當作命令執行時,爲安全起見,應先調用 shellescape() 或 fnameescape() 進行轉義。這兩個函數自有其轉義策略，適用大部分情況。當有特殊需求時，可用 escape() 指定要轉義哪些字符（傳入第二參數的字符串中出現的所有字符都表示要轉義的)。

• tolower() 將字符中轉爲小寫

• toupper() 將字符串轉爲大寫

• tr() 按一一對應的方式轉換字符串

• strtrans() 將字符串轉換爲可打印字符串

tr() 函數進行簡單的字符串轉換（不是正則替換)，效果如同 unix 工具 tr。大小寫轉換是其一種特例策略，如轉大寫相當於 tr(str, 'abcdefg...', 'ABCDEFG...')。而 strtrans() 是按 vim 的自定策略將不可打印字符轉換爲可視字符（組合，一般以 ^ 開頭）表示。

• strlen() 按字節數獲取字符串長度

• strchars() 按字符數獲取字符串長度，當含寬字符（如漢字時）與字節長度有差異

• strwidth() 字符串寬度，顯示在屏幕上時將佔用的列寬度，但未處理製表符

• strdisplaywidth() 字符串實際顯示寬度，並按設置處理製表符寬度

• byteidx() 第幾個字符的字節索引，不單獨處理組合字符

• byteidxcomp() 也是字符索引轉爲字節索引，組合字符單獨處理

字符串可像列表一樣用中括號索引,那是按字節索引的。當字符串中存在寬字符時,字符數與字節數不一致，這就需要處理字符索引與字節索引的不同。請仔細觀察以下示例:

```
: let str = 'vim 中國'
: echo strlen(str)    |" --> 10
: echo strchars(str)  |" --> 6 (vim 加空格加兩漢字)
: echo len(str)       |" --> 10
: echo strwidth(str)  |" --> 8 (每個漢字三字節但兩寬度)
: echo str[0:2]       |" --> vim
: echo str[4]         |" --> <e4> (中字的第一個字節)
: echo str[4:5]       |" --> <e4><b8>
```

```
: echo str[4:6]       |" --> 中
```

組合字符（有的書籍叫重音字符），中國人一般不必關注，歐洲人才用得到。比如 é 是通過一個正常的 e 字母加上重音符組成而成的（e . nr2char(0x301)），顯示上像是一個字符，但計算機要用兩個字符表示。至於算一個字符還是兩個字符，似乎都有理有據，所以就提供了不同的函數或可選參數來處理這種情況。這與漢字寬字符的情況不一樣。漢字的三個字節是不可分的，取第一字節是無效字符。但組合字符的第一字節仍是個有效字符（字母）。

字符與編碼看似簡單，如同空氣與水一樣簡單，但深入細處還挺複雜。所以建議初學者不必深究，始終用英文文本示例測試學習即可。當實際工作中遇到中文問題時再回頭查閱。此外，據說早期的中國程序員常要念經 “一個漢字等於兩個字節”，那是用 GB 編碼的原因，現在請升級經文 “一個漢字等於三個字節”。

- stridx() 查找一個短字符串在另一個長字符串第一次出現的起始索引
- strridx() 查找一個短字符串在另一個長字符串最後一次出現的起始索引
- strpart() 截取字符串從某個索引開始的定長子串

查找簡單子串存在情況可用 stridx() ，要求精確匹配，且大小寫敏感。返回的結果索引是字節索引，索引從 0 開始，若不存在子串返回 -1。截取子串可用中括號索引切片方式，如上例 str[4:6]，參數是起始索引與終止索引（含雙端）。而 strpart() 的參數是起始索引與長度，如上例等效於 strpart(str, 4, 3)。

- match() 查找一個正則表達式在字符串出現的起始索引，不匹配時返回 -1
- matchend() 查找一個正則表達式在字符串出現的終止索引
- matchstr() 返回字符串中匹配正則表達式的部分，不匹配時返回空串
- matchlist() 將正則匹配結果按分組返回至列表中，不匹配時返回空列表
- substitute() 正則表達式替換，‘:s’ 命令的函數式
- submatch() 獲取正則匹配的分組子串，只可用於 ‘:s’ 命令的替換部分

這幾個函數用於處理正式表達式的匹配查找與替換。如果僅是要判斷是否匹配，可直接用操作符 if str =~# pattern。match() 函數主要是還能返回匹配成功的起始索引，相應地 matchend() 返回的是終止索引。matchstr() 返回的是匹配到的整個子串。如果正則表達式中有括號分組 ，最好用 matchlist() 函數，它返回的列表中，第一個元素（[0]）就是匹配到的整個子串，其後是按順序的分組子串。其中的關係可用如下僞代碼表示：

```
let s = some_string
let p = search_pattern
if some_string =~# search_pattern
    let sidx = match(s, p)
    let eidx = match(s, p)
    sidx != -1; eidx != -1
    s[sidx:eidx] == mathcstr(s, p)
    let slist = matchlist(s, p)
```

```
 9      slist[0] == matchstr(s, p) == & == submatch(0)
10      slist[1] == \1 == submatch(1)
11      slist[2] == \2 == submatch(2)
12      ...
13  endif
```

替換函數 substitute() 的參數及意義與 ':substitute' 命令的幾個部分完全一樣。不過命令可以縮寫爲 ':s', 函數不可以縮寫。如以下兩個語句功能類似:

```
: s/pat/sub/flag |" 對當前行改 pat 爲 sub
: call substitute(line('.'), pat, sub, flag)
```

在替換部分 {sub} 可用表達式, 以 \= 開始即可, 如此 submatch() 表示前面 pat} 部分的分組子串。而在以常規字面字符串表示 {sub} 部分時, 則用 \1 表示分組子串。

- string() 將其他任意變量或表達式轉爲字符串表達, 類似 ':echo' 的顯示
- expand() 將具有特殊意義的標記 (如%, #, <cword> 等) 展開
- iconv() 轉換字符串編碼
- repeat() 將字符串重複串接多次生成長字符串
- eval() 將字符串當作表達式來執行, 並返回結果
- execute() 將字符串當作命令來執行, 將結果返回爲字符串

注意, eval() 與 execute() 很靈活, 但比較低效, 也可能有一定風險。如有其他更優雅的實現寫法, 儘量用替代方案。

浮點數學運算

用 VimL 做數學運算並不常見, 但如果啥時想到需要她, 她也在那兒。一般整數運算直接用操作符, 浮點運算才需要調用函數, 且這些內置函數的結果一般也是浮點數, 即使參數都是整數。

- float2nr() 將浮點數轉爲整數類型
- trunc() 截斷取整
- round() 四舍五入取整
- floor() 向下取整
- ceil() 向上取整

這幾個函數都是取整運算,但實際上只有 'float2nr()' 的結果是整數類型(v:t_number), 其他函數取整後仍爲浮點數 (v:t_float)。float2nr() 與 trunc() 意義一樣是截斷取整。當涉及負數, 取整可能不太直觀, 請看示例:

```
: echo float2nr(4.56) float2nr(-4.56) |" --> 4 -4
: echo trunc(4.56) trunc(-4.56)   |" --> 4.0 -4.0
: echo round(4.56) round(-4.56)   |" --> 5.0 -5.0
: echo floor(4.56) floor(-4.56)   |" --> 4.0 -5.0
```

```
: echo ceil(4.56) ceil(-4.56)     |" --> 5.0 -4.0
```

當四舍五入正好在中值時（如小數部分是 0.5），取遠離 0 那個整數，類似：

```
: round(+float) == trunc(+float + 0.5) |" --> 正數取整
: round(-float) == trunc(-float - 0.5) |" --> 負數取整
```

- fmod() 取余數
- pow() 取冪
- sqrt() 開平方

整數取余數可直接用操作符% ，該操作符不能用於浮點數。fmod() 可用於浮點數的取余，即使整數也當作浮點處理。VimL 並沒有整數取冪的操作符（其他語言有用 ^ 或 ** 作冪運算的），須用 pow() 函數求冪，結果也總是浮點數；

```
echo 10 % 3          |" --> 1
echo fmod(10, 3)     |" --> 1.0
echo pow(2, 10)      |" --> 1024.0
echo pow(4, 1/2)     |" --> 1.0  (先計算 1/2 = 0)
echo pow(4, 1/2.0)   |" --> 2.0
echo sqrt(4)         |" --> 2.0
```

- exp() 自然指數
- log() 自然對數，以 e = 2.718282 爲底
- log10() 常用對數，以 10 爲底
- 三角函數與反三角函數：sin()，cos()，asin()，acos() 等

log() 是 exp() 的反函數，在數學上的記號是 ln；而數學上記爲 lg 的對數，程序上是 log10()。這個命令習慣應該是源自 C 語言的標准庫函數。同樣，一衆三角函數也是類似 C 語言的，參數是以弧度單位表示的角度。不過很難想像需要在 VimL 中用到這些略爲高深的數學計算的場景。

- isnan() 判斷是否爲非數

自 Vim8 引入非數的判斷。像 0/0 這樣的計算結果叫非數，在其他一些計算機語言與文檔中習慣用 NaN 來表示。可能爲了更好地與其他數據文件交互，Vim 也增加這個函數來處理非數。

列表與字典運算

在第四章介紹數據結構時，已經順便介紹了操作列表與字典的函數。這裏不再重複，只作些補充說明。

首先，很多函數可同時作用於列表與字典，甚至字符串。因爲腳本語言弱類型的緣故，沒法限定傳入函數的參數，只能根據參數類型作出不同的合理反饋。例如：

- len() 取列表或字典集合中元素個數，也取字符串的（字節）長度

• empty() 可判斷是否空列表、空字典或空字符串，整數 0 也認爲是空的

• match() 還能匹配字符串列表，返回能匹配成功的元素索引

其次，列表與字典都是集合，有一類高階函數，可接收另一個函數（引用）作爲參數，用於處理集合內的每一個元素。比如 map() 與 filter() 函數。

前面提及，VimL 有許多與命令同名的函數，都是實現類似的功能。但 map() 是例外，它與定義鍵映射的‘:map’命令沒有語義關係，完全是不同的概念。

• map({expr1}, {expr2}) 修改集合的每個元素

其中參數一 {expr1} 可以是列表如字典，參數二 {expr2} 是函數引用。參數一集合的每個元素，傳給參數二所代表的函數，將結果值替換原來的元素。最終會原位修改列表或字典。關鍵是參數二所引用的函數定義要遵循一定的規範，它應接收兩個參數，map() 會將每個元素的索引與值傳給該函數（字典元素的索引即是鍵名）。整個流程可如下模擬：

```
function! MapDict(dict, fun)
    for [l:key, l:val] in items(a:dict)
        let l:val_new = a:fun(l:key, l:val)
        let a:dict[l:key] = l:val_new
    endfor
endfunction

function! MapList(list, fun)
    for l:idx in range(len(a:list))
        let l:val = a:list[l:idx]
        let l:val_new = a:fun(l:idx, l:val)
        let a:list[l:idx] = l:val_new
    endfor
endfunction
```

如果處理每個元素的函數很簡單，則可不必創建函數再傳入函數引用。可用一個字符串代替，該字符串調用 eval() 執行後，將結果替換原元素。在字符串中，用內置變量 v:val 代表迭代的每個元素值，v:key 代表元素索引（鍵名）。相當於傳入函數版本的兩個參數。用這種方法得注意字符串的轉義，建議用單引號括起字面字符串。可用其他字符串函數或操作符組裝，最終結果的字符串再調用 eval() 計算結果新值。

事實上，在低版本的 vim 中，map() 函數的參數二隻能用字符串。這才需要 v:key 與 v:val 這兩個特殊變量標記置於可執行字符串中。自 vim8 後，強烈建議使用函數引用參數。這就無須理解 v:key 與 v:val 的即時意義。不過在定義處理元素的函數時，建議也用 key 與 val 作爲函數形參，這使整個代碼的可讀性更佳：

```
function! MapHandle(key, val) abort
    let l:result = deal with a:key and a:val
    return l:result
```

```
4 endfunction
```

如果處理函數很簡單，也可不必預定義函數，即時定義 lambda 也可以，因爲 lambda 表達式的值也正是一個函數引用。例如：

```
let list1 = [1, 2, 3]
let list2 = map(list2, {idx, val -> val * 2})
echo list1
echo list2

let list3 = map(copy(list2), {idx, val -> val * 3})
echo list2
echo list3
```

以上示例也說明了 map() 函數是原位修改的，如果不想修改原集合，可先調用 copy() 創建副本。低版本中等效的用字符串調用方式如下：

```
echo map([1, 2, 3], 'v:val * 2')
```

像這樣簡單的功能，也許字符串方式寫來更簡潔，但稍爲複雜的功能，可執行字符串的表示法就可能比較費解了。比如要將原列表中每個元素加上尖括號 <> 括起來，以下三種調用方式都能實現：

```
echo map([1, 2, 3], '"<" . v:val . ">"')
echo map([1, 2, 3], 'printf("<%s>", v:val)')
echo map([1, 2, 3], {idx, val -> printf('<%s>', val)})
```

用 lambda 表達式可避免多重引號的理解困難。而且 lambda 表達式或函數若預先定義的話，在其他地方也是可用的。而含 v:val 的特徵字符串，放在其他地方几乎是沒什麼意義了。

• filter({expr1}, {expr2}) 過濾集合內的元素

filter() 與 map() 函數類似。參數二所代表的處理函數，也接收索引與值兩個參數，但是要求返回布爾邏輯值。如果返回的是真（數字 1），則保留不處理，如果返回的是假（數字 0），則刪除相應的元素。模擬流程如下：

```
1 function! FilterDict(dict, fun)
2     for [l:key, l:val] in items(a:dict)
3         let l:bKeep = a:fun(l:key, l:val)
4         if empty(l:bKeep)
5             unlet a:dict[l:key]
6         endif
7     endfor
```

```
8 endfunction
```

自己模擬過濾列表可能略有麻煩，因爲如果正向迭代，刪除元素後，索引可能會變化。當然了，你不必真的自己寫或用這樣的模擬函數，請用內置的庫函數！

同樣地，可以用字符串或 lambda 表達式。且在支持 lambda 表達式的 vim 中，儘量用 lambda 表達式。

- sort({list} [, {fun}, {self}]) 爲列表排序，從小到大
- uniq({list} [, {fun}, {self}]) 刪除相鄰重複元素

幾乎在任一本算法教科書，排序都是重點。但是幾乎在任一個語言中，排序都有已優化實現的庫函數，不必自己寫的，自己需要做的只是提供比較函數，說明要如何排序的需求。VimL 要求的比較函數能接收兩個參數，返回值意義如下：

- 0 兩個參數視爲相等
- 1 第一個參數視爲比第二個參數大
- -1 第一個參數視爲比第二個參數小

sort() 函數只能爲列表排序，因爲字典是無序的。第二參數 {fun} 一般是函數引用，可用 lambda 表達式，但不支持像 map() 那樣的可執行字符串。然而，可以是普通字符用於表示 vim 預設的幾種排序策略（常用需求）：

- 空串或省略，按字符串排序，類似‘:sort’命令爲當前 buffer 的排序行爲。
- l 或 i，忽略大小寫的排序
- n 按數字排序，非數字類型的元素認爲是 0
- N 按數字排序，字符串會轉爲數字
- f 按數字排序，列表元素限定僅是數字或浮點數

VimL 的 sort() 是穩定排序算法，即如果兩個元素相等（按 {fun} 返回 0），排序後它們也保持原來的相對順序。如果 {fun} 參數是含 dict 屬性的函數，則要提供第三參數 {self}，一個作爲 self 的字典變量。

uniq() 函數的參數用法與 sort() 相同。且一般應該對已排序的列表調用 uniq()，因爲它只比較相鄰元素而去重。

小結

VimL 的標量主要就是字符串與數字，集合也就列表與字典。所以爲這些數據類型提供了大量的庫函數 api。用‘:h type()’查看支持的所有變量類型。但其他類型需要支持的操作非常有限，故無必要有什麼專門函數處理。

6.2 操作編輯對象

與 Vim 可視編輯的有關的幾個概念對象是緩衝（buffer）、窗口（window）與標籤頁（tabpage），還有目前較少用到的在命令行參數提供的文件列表（argument list）。VimL 也提供了許多函數以供腳本來控制這些編輯對象。

編輯對象背景知識

很早期的 vi 一次只能編輯一個文件。不過從命令行啓動時可以提供多個文件名參數，首先編輯第一個文件，編輯完後可以接着編輯下一個文件。如以下命令啓動：

```
$ vim file1 file2 file3
```

Vim 就記憶着這三個文件，稱之爲參數列表，相當於執行了如下 VimL 語句：

```
: let arglist = ['file1', 'file2', 'file3']
```

注意在 Vim 啓動時，還可以加很多命令行選項（以 '-' 開頭的參數），一般用於指定 Vim 以何種方式、何種配置等啓動。這些選項在 Vim 啓動過程中就會被處理掉，不會保存在參數列表中，所以參數列表只保存待編輯的文件名。

後來，Vim 支持同時編輯多個文件。作爲通用編輯器，配置好 vimrc 後，它也經常省略命令行參數，直接以裸命令 $ vim 啓動，其參數列表就爲空 []。然後在 Vim 自己的命令行中用命令 ':edit file' 打開要編輯的文件。

Vim 每打開一個文件，就創建一個緩衝 (buffer)，並記錄相應的緩衝信息。即使打開另一個文件，曾經打開的而目前看不見的文件，也記憶着它的緩衝，除非用命令顯示地清除它。Vim 的這個緩衝概念與系統緩存並不一樣，對於非活躍 buffer（看不見的文件），Vim 也不可能將文件的所有內容留在內存中，尤其是打開了很多大文件。可認爲 buffer 是 Vim 爲每個編輯文件創建的一個對象，記錄着一些必要的信息。但是，也不一定每個 buffer 都對應着文件系統內的物理文件（磁盤上的文件），例如新建 buffer 尚未保存甚至未命名，還有很多標准插件與三方插件的輔助窗口中的特殊 buffer 根本就不想寫入文件。

然後正在編輯的活躍 buffer 必然是顯示在窗口的。早期的 vi/vim 也只支持一個窗口，後來實現了多窗口。一個窗口只能裝載一個 buffer，但一個 buffer 可以同時顯示在多個窗口中。再後來更擴展到多個標籤頁，每個標籤頁都可以分隔爲多個窗口。

緩衝、窗口與標籤頁都被 Vim 順序編號以便維護，這有點像參數文件列表的索引。不過參數列表是真當作列表變量類型的，索引從 0 開始。而緩衝、窗口與標籤頁的編號都從 1 開始。關閉文件並不意味着關閉緩衝，即使清除緩衝或隱藏緩衝也不會改變每個緩衝的編號，但是關閉或移動窗口（或標籤頁），卻會改變它們的編號。爲此，自 Vim8 起，又引入窗口 id 概念，它是唯一且穩定的（不過似乎尚未有標籤頁 id 的概念）。

然而要指出，即使引入了緩衝概念，參數列表也還是有價值的。在有些情況下啓動 vim 確實有明確目標要編輯某系列文件，將所有文件保存在參數列表中（其實在進入 vim 後也可以提供或更改文件參數列表），就有很多批量命令能統一處理這系列文件。即使引入窗口 id 概念，也還有窗口編號的價值。因爲窗口編號更直觀，從左到右，從上到下，很容易知道哪個窗口是 1、2、3、4。

查看緩衝可用如下命令之一：

```
: buffers
: ls
```

注意 ':buffers' 是有 s 後綴的複數形式，那纔是打印緩衝列表的意思。如果是單數命令 ':buffer' 則一般需要接個參數，用於打開另一個緩衝的意思。':ls' 更簡短，這命令在 shell 中是列出文件意思，而在 Vim 中是列出緩衝的意思。這兩個命令的輸出中，包含緩衝編號及相應的文件名等信息。

在任一時刻，都有（正在編輯中的）當前緩衝，當前窗口與當前標籤頁的概念。如果提供了參數文件列表，也有當前文件的概念。不過當前文件不一定與當前緩衝相同。因爲常規編輯命令不會改變參數列表，你可以用 ':e' 或 ':b' 命令切換到編輯另一個可能並不在參數列表中的 "無關" 文件，但在 vim 內部隨參數文件列表保存的當前索引並不會改變。

最後，將這三個或四個概念統稱爲編輯對象。當了解這些編輯對象的意義後，就能更好地理解相關的函數功能了。初學者可能會對緩衝與文件（參數列表）有所迷惑，日常使用時可不求甚解認爲緩衝即是指文件。不過編程時需要准確理解其中的不同。

獲取編輯對象信息

- bufnr() 獲取緩衝編號
- bufname() 獲取緩衝名字
- winnr() 當前窗口編號，winnr('$') 獲取窗口數量即最大編號
- tabpagenr() 當前標頁面編號，'$' 參數獲取標籤頁數量
- tabpagewinnr() 某個標籤頁的當前窗口編號
- bufwinnr() 獲取某個緩衝的窗口編號
- winbufnr() 獲取某個窗口的緩衝編號

其中，bufnr() 與 bufname() 的參數是一樣意義，指示如何搜索一個緩衝，搜索失敗時前者返回 '-1'，後者返回空字符串：

- 數字：即表示緩衝編號。bufnr(nr) 一般返回編號本身（無效時返回 '-1'）。bufname(nr) 用於獲取指定編號的緩衝文件名。
- 字符串：除了以下特殊字符意義，將該字符串當作文件名模式去搜索緩衝，也就是說不必指定文件全名，可以按文件名通配符（不同於正則表達式）搜索。但如果有歧義能匹配多個，或未能匹配，都算失敗，返回 '-1' 或空串。當然會優先匹配全名，如果要限定只當作全名匹配，可加前後綴 '^' 與 '$'。
- 缺省：不能缺省參數，至少提供空字符串。
- "": 空字符串表示當前緩衝。
- "%": 也表示當前緩衝。
- "#": 表示另一個輪換緩衝（在編輯當前緩衝之前的那個緩衝）。
- 0：數字零也表示另一個緩衝。

注意，bufname() 返回的緩衝名，與 ':ls' 命令輸出的相應緩衝行的主體部分相同。該緩衝名是否包含文件全路徑名，可能與當前路徑有關。所以，如果要在程序中唯一確定一個緩衝，應該用 bufnr() 的返回值，bufname() 一般只用於顯示。

bufnr() 函數不能無參數調用，空字符串或數字零都是有特殊意義的參數。但是，winnr() 與 tabpagenr() 一般是無參數調用，以獲取當前窗口（標籤頁）的編號，而用"$" 參數表示獲

取最後一後窗口（標籤頁）編號，也就是最大編號或其總數量。tabpagewinnr() 用於獲取另一個標籤頁的當前窗口編號，比 winnr() 多加一個標籤頁編號參數在前面。因爲每個標籤頁都有當前窗口的概念，即是最後駐留的那個窗口。此外，"#" 參數可用於 winnr() 與 tabpagewinnr() 表示之前窗口編號（即進入當前窗口之前的那個窗口，<C-w>p 或 ':wincmd p' 將進入的窗口）；但不可用於 tabpagenr() 函數，因爲 Vim 似乎沒有維護之前標籤頁的概念。

以窗口編號爲例，其典型調用方式小結如下：

- winnr() 當前窗口編號
- winnr('$') 最大窗口編號或窗口數量
- winnr('#') 之前窗口的編號

因爲一個緩衝可能顯示在多個窗口中，所以 bufwinnr() 返回的是顯示了指定緩衝的第一個窗口編號。其參數與 bufnr() 意義相同，可認爲先調用 bufnr() 確定緩衝編號再查找相應窗口編號。反之，一個窗口在一個時刻只顯示一個緩衝，所以 winbufnr() 返回的緩衝編號是確定的。其參數是窗口編號，用 0 表示當前窗口，但不能像 winnr() 那樣使用 $ 或 # 字符表示特殊窗口，否則字符串按 VimL 自動轉換規則轉爲數字 0，仍是調用 winbufnr(0)。

- bufexists() 檢測一個緩衝是否存在
- buflisted() 檢測一個緩衝是否能列表出來（:ls）
- bufloaded() 檢測一個緩衝是否已加載
- tabpagebuflist() 返回顯示在某個標籤頁中的所有緩衝編號列表

以上三個檢測緩衝狀態的函數，所有接收的參數除了緩衝編號外，若字符必須是文件全名（全路徑或相對當前路徑），並能像 bufnr() 的參數那樣支持文件通配符。一些特殊緩衝並不會被列表出來，取決於局部選項 &buflisted 的設置。已加載的緩衝是指顯示在某個窗口的緩衝，但如果一個緩衝設置了 &bufhidden 局部選項爲可隱藏 hide，則它即使不顯示了也仍算加載狀態。

若要獲取所有已顯示在窗口中的緩衝，可用 tabpagebuflist() 函數，它返回一個列表，收集了指定標籤頁中所有窗口內顯示的緩衝（編號）；缺省參數是指當前標籤頁。在所有標籤頁中顯示的緩衝都是已加載狀態（但已加載緩衝可能還包含一些隱藏緩衝），如下函數可返回幾乎所有已加載緩衝的列表：

```
1 function! BufLoaded() abort
2     let l:lsBufShow = []
3     for i in range(1, tabpagenr('$'))
4           call extend(l:lsBufShow, tabpagebuflist(i))
5     endfor
6     return l:lsBufShow
7 endfunction
```

- argc() 參數文件列表個數
- argv() 參數文件列表，或返回指定索引的文件參數

- argidx() 當前所處參數文件的索引
- arglistid() 返回參數文件列表的 ID

這是幾個處理參數文件列表的函數。在去除啓動選項後，argc() 與 argv() 就是命令行參數。無參數調用 argv() 返回整個文件列表，但可指定索引 argv(idx) 返回相應的文件名，argc() 就是這個列表的長度，即文件個數，而 argidx() 是指所謂的當前文件的索引。但是 Vim 還對參數文件列表作了擴展，除了從命令行啓動時指定的參數列表叫做全局參數文件列表外，還可以爲每個窗口定義局部參數文件列表，所以有了 arglistid(winnr, tabnr) 函數用以返回某個指定窗口（參數都可缺省，即用當前窗口或當前標籤頁）的參數文件列表，全局的參數文件列表 ID 用 0 表示。

- win_getid() 獲取指定標籤頁與窗口編號（可缺省默認當前）的窗口 ID
- win_gotoid() 切換到指定窗口 ID 的窗口，有可能切換當前標籤頁
- win_id2win() 將窗口 ID 轉換爲窗口編號，只在本標籤頁查找
- win_id2tabwin() 將窗口 ID 轉換爲二元組 [標籤頁編號, 窗口編號]
- win_findbuf() 根據緩衝編號查找所有相應的窗口 ID（是列表類型）

這幾個處理 window-ID 的函數是從 Vim8 版本引入的。函數名已經很望文生義了，可以在窗口 ID 與窗口編號（及標籤頁編號）之間互相轉換。要注意的是，每個標籤頁的窗口編號都是從 1 開始重新編號，相互獨立。但窗口 ID 是全局的，所有標籤頁的窗口共享一套統一的 ID。

獲取編輯對象數據

前文在介紹 VimL 變量作用域時，提到三個特殊的局部作用域前綴 'b:', 'w:', 't:'，那就是分別保存在特定緩衝、窗口與標籤頁的變量。如果僅用這個前綴，而無後綴主體變量名，那就是表示收集了所有相應局部變量的字典（如 'b:' 也是個類似 's:' 的特殊字典）。從語義上理解，字典可當作一個對象，鍵當作屬性。那麼這些局部變量也就相當於相應編輯對象的屬性數據了。以下的 get/set 函數就是處理這些變量的函數：

- getbufvar() 返回緩衝局部變量 'b:'
- setbufvar() 設置緩衝局部變量的值
- getwinvar() 返回窗口局部變量 'w:' （限當前標籤頁）
- setwinvar() 設置窗口局部變量的值
- gettabvar() 返回標籤頁局部變量 't:'
- settabvar() 設置標籤頁局部變量的值
- gettabwinvar() 返回窗口局部變量 'w:'
- settabwinvar() 設置窗口局部變量的值

以緩衝局部變量爲例，函數參數原型是 getbufvar(緩衝, 變量名, 默認值)。其參數一是緩衝編號或名字（類似 bufnr() 的參數意義）；參數二的變量名是沒有 'b:' 前綴的主體名字，即 'b:' 字典的鍵；參數三是默認值，當不存在相應變量時的返回值，該參數可缺省，缺省時就是空字符串，即當變量不存在時也不會出錯，而至少返回空字符串。參數二變量名不

可缺省，當它是空（字符串）時，返回 'b:' 字典本身。設值函數參數原型時 setbufvar(緩衝，變量名，新值)，第三參數不可缺省。

窗口局部變量取值與設值函數，可能與標籤頁有關。gettabwinvar(標籤頁號，窗口編號，變量名，默認值)，需要在第一個參數前多插入一個標籤頁編號，如果取當前標籤頁的窗口變量，則用 getwinvar(窗口編號，變量名，默認值)。窗口編號參數傳 0 的話，表示當前窗口。

- getbufinfo() 返回緩衝對象信息列表
- getwininfo() 返回窗口對象信息列表
- gettabinfo() 返回窗口對象信息列表

這三個函數是從 Vim8 引入的。其返回類型是字典的列表，即每個列表元素都是字典，字典所包含的屬性鍵依對象而不同。如果參數限定了一個對象，返回值也是包含一個元素的列表；如果根據參數無法確定（搜索到）任一對象，則返回空列表。如果沒有參數，則返回由所有對象的信息字典組成的列表。

如果提供參數，getwininfo() 需傳入窗口 ID，而 gettabinfo() 傳入標籤頁編號。而 getbufinfo() 稍爲複雜，除了可像 bufnr() 那樣傳入緩衝編號或名字外，還可以用字典指定篩選緩衝的條件：buflisted 已列出的，bufloaded 已加載的。

這三個函數返回的對象信息字典，詳細的鍵名解釋請參考文檔。但是都有一個鍵 variables （注意單詞複數形式），其值是另一個字典（引用），即是特殊字典 'b:' 或 'w:' 't:'。所以 get...info() 函數也實現了 get...var() 的功能，不過前者所得信息大而全，用法更復雜。另外 get...var() 函數可獲取局部選項的值，以 & 爲前綴的變量名傳入即可，但這無法由 get...info() 獲得，因爲選項值並不保存在 'b:' 字典中。

獲取光標位置信息

顯然，當前光標只有一個確定位置。但 Vim 另有一個光標標記（mark）的概念，用於記憶多個位置信息。例如在普通模式下用 mx 命令，就定義了標記 x ，保存着當前光標的位置。此後移動到他處後，再用命令 'x（單引用）就能跳回標記 x 的行首，使用'x （反引號）就跳回標記 x 的准確行列位置。每個緩衝都能讓用戶定義以小寫字母 a-z 爲名的標記，稱爲局部標記；而大寫字母爲名的標記是全局的，可以跨文件緩衝跳轉。此外，Vim 還有些自動定義的標記，如在選擇模式下按 ':' 進入命令行，會自動添加 ':'<,'>' ，那就分別表示選區起始行與終止行的標記。

- line() 光標或標記的行號
- col() 光標或標記的列號（字節索引）
- virtcol() 光標或標記的屏幕佔位列號
- winline() 光標在當前窗口的行號
- wincol() 光標在當前窗口的列號
- screenrow() 光標在屏幕的行號
- screencol() 光標在屏幕的列號

以上 line()，col() 返回的行列號是相當緩衝文件而言。col() 是按字節列號的，第一列是 1，0 用於表示錯誤列號。virtcol() 指屏幕佔位列號，光標所在字符所佔的最後一列。假如一行全是漢字，光標停在第四個漢字上，col() 是 10，因爲前三個漢字只 9 字節，第四漢字從第 10 字節開始；virtcol() 是 8 ，因爲每個漢字佔兩列寬，第四個漢字已佔到第 8 列。當有製表符 ^I 屏幕列與字節顯然也是不同的。不過這三個函數必須帶參數調用，字符串參數意義如下：

- . 單點號表示當前光標
- $ 當前行最後一列
- 'x 表示 x 標記
- v 用於選擇模式下，表示選區起始（因當前光標只表示選區終止）

特殊用法是 col([行號, '$']) 可獲得指定行的最後一列。

winline() 與 wincol() 不帶參數，只用於獲取當前光標相對於窗口的行列號，因爲標記位置可能不在窗口顯示區域，爲標記調用這兩個函數無意義。winline() 與 line('.') 的意義不同顯而易見，長文件經常滾動，窗口的第一行在不同時刻對應着文件的不同行。水平滾動條不如垂直滾動條用得多，但即使無水平滾動，wincol() 可能也與 col('.') 不同。仍以上例漢字行，光標停在第四漢字上，wincol() 返回的是 7，因爲前三漢字佔 6 屏幕寬度，第四字從第 7 開始。

因爲 Vim 可以分隔多個窗口，所以屏幕行列號 screenrow()，screencol() 又與窗口行列號 winline()，wincol() 不同。不過屏幕行列號一般只用於測試。且直接在命令行手動輸入 ':echo screencol()' 時，它始返回 1 ，因爲執行命令時光標已經在命令行首列了。

- getpos() 獲取光標或標記的位置信息
- setpos() 設定光標或標記的位置信息
- getcurpos() 獲取當前光標的位置信息
- cursor() 放置當前光標

顧名思義，可能會覺得 getpos() 就是 line() 與 col() 的綜合效果，但其實位置信息不僅是行列號。'getpos() 的返回值是一個四元列表 [bufnr, line, col, off]，其意義如下：

- bufnr 緩衝編號，0 表示當前緩衝，只有在取跨文件的全局標記，才需要返回其所在緩衝的編號，否則就是 0。
- line 行號，這就相當於 line() 函數了
- col 列號，這就相當於 col() 函數了
- off 偏移，只有在 &virtualedit 選項打開時纔不是 0。比如 <Tab> 鍵可能佔多列，但在一般情況下移動光標時是直接跳過的，但在打開 &virtualedit 選項時，就可能移動到製表符中間某個位置了，這就是第四個返回值的意義。

setpos() 是 getpos() 的對應函數，它所接收的第二參數就是後者返回的四元列表。第一參數就是標記名 'x（注意含單引用，而非反引號）或表示當前光標的 '.'。

getcurpos() 無參數，只返回當前光標的位置信息，基本與 getpos('.') 功能相同，不過返回值列表還多一個第五元素 curswant ，它表示當光標垂直移動（jk）時，它優先移動到的列號，因爲當前列號在下一行或上一行未必是有效的，這時該移動到哪列呢，這第五個

返回值就有效果了。

cursor() 用於放置當前光標，從語義上是 getcurpos() 函數的“反函數”，但是卻不能將後者的返回參數傳給前者。因爲 getcurpos() 返回值是五元列表，而 curosr() 函數用不到其第一個返回值 bufnr，將第一個元素移除後的列表傳給 cursor() 是可行的。事實上，cursor() 還可以幾個非列表的參數直接調用。如 cursor(line, col, off) ，或 coursor([line, col, off, curswant]) 當然，只有行列號是必須的。當需要明確移動光標到某處時，直接調用 cursor(line, col) 是最方便的。當需要恢復光標時，最好與 setpos() 聯用，如：

```
: let save_cursor = getcurpos()
" 移動光標干活
: call setpos('.', save_cursor)
```

- byte2line() 文件的第幾字節處於第幾行
- line2byte() 第幾行是從文件第幾字節開始的

這兩個函數將整個緩衝文件的字節索引與行號相互轉換。注意包含換行符，換行符是一字節還是兩字節則與文件格式有關。line2byte(line(‘$’) + 1) 可獲取緩衝的大小，其實比緩衝大小多 1，因爲是文件最後一行的下一行的起始索引。除此之外，非法行號返回‘-1’。

- winheight() 返回指定編號窗口的高度，參數 0 表示當前窗口
- winwidth() 返回指定編號窗口的寬度
- winrestcmd() 返回一系列可恢復窗口大小的命令
- winsaveview() 保存當前窗口視圖，返回一個字典
- winrestview() 由保存的字典恢復當前窗口視圖

注意，winrestcmd() 只能恢復窗口大小，以字符串形式返回，將它用於‘:execute’執行後才能恢復窗口大小。而 winsaveview() 與 winrestview() 能保存恢復比較完整的窗口信息。其參數字典保存哪些鍵名及釋義請參閱相關文檔。

- screenchar() 返回屏幕指定行列座標的字符
- screenattr() 返回屏幕指定行列座標的字符有關的特徵屬性

Vim 的屏幕不僅包括緩衝窗口，還有標籤頁行，狀態欄，命令行，窗口分隔符等都佔據一定屏幕座標。不過這兩個函數主要用於測試。

操作當前緩衝文本

然後是操作緩衝文件文本內容的函數，這是 Vim 作爲文本編輯器的基礎工作。

- getline() 從當前緩衝中獲取一行文本字符串，或多行組成的列表
- setline() 從當前緩衝指定行開始替換文本行
- append() 從當前緩衝指定行下方始插入文本行
- getbufline() 從指定緩衝中獲取文本行
- wordcount() 統計當前緩衝的字節、字符、單詞，返回值是字典

如果 getline() 傳入一個行地址參數，則返回一個字符串；如果傳入兩個起止行地址參數，則返回一個列表，每個元素爲一行文本。行地址參數可以是數量或字符 '.' 表示當前行，字符 $ 表示最後一行。setline() 可傳入一個行地址參數，以及一個字符串或字符串列表，用以替換指定行以及後續行。append() 用法與 setline() 一樣，不過是從指定行（下方）開始插入，並不會覆蓋原有行。

getbufline() 與 getline() 類似，不過是取其他緩衝，所以要在第一個參數多傳入一個緩衝編號或名字。另外，行地址參數不能用 '.' 點號表示當前行，因爲在其他緩衝的當前行意義不明顯（用戶角度），而且返回值必定是列表，即使只有一個起始行地址參數，也是一個元素的列表。

- mode() 當前的編輯模式：普通、選擇、命令行等
- visualmode() 上次使用的選擇模式：字符、行、或列塊選擇

Vim 有很多種模式，在腳本中可用這兩個函數獲取模式信息，然後根據模式作不同的響應工作。一個非常有用的用途是用於狀態欄定製中，否則觸發該函數的時刻經常是命令行模式（通過命令行調用或加載腳本），或普通模式（映射中調用）。

- indent() 指定行的縮進空白列數
- cindent() 按 C 語法應該縮進的空白列數
- lispindent() 按 Lisp 語法應該縮進的空白的列數
- shiftwidth() 每層縮進的有效空白列數

這幾個縮進函數其實都是隻讀函數，並不會改變緩衝內容（執行縮進操作的命令 =）。indent() 是返回指定行（參數按 getline() 慣例）的當前實際縮進數，按縮進的空白數計，如果縮進字符是製表符，與相關的製表符寬度選項有關。而 cindent() 與 listindent() 是假設按 C 或 Lisp 語法規則縮進，該行應該縮進多少。需要根據這個返回結果調用其他命令或函數執行真正的修改操作。與縮進相關的選項有好幾個，而 shiftwidth() 函數是綜合這幾個選項的設置，給出的當前緩衝實際生效的每級縮進數量。

- nextnonblank() 尋找下一行非空行
- prevnonblank() 尋找上一行非空行

這兩個函數很簡單，就是從參數指定的起始行地址查找非空行，如果起始行已經是非空行，直接返回該行地址。返回值是數字，失敗時返回 0，因爲行地址索引從 1 開始。作爲通用文本編輯器，Vim 假定文本文件用空行分隔段落。而且良好編程風格的大多數語言源文件，也是應該有空行分隔段落的，所以這兩個函數有時挺實用。

- search() 搜索正則表達，返回行地址
- searchpos() 搜索正則表達式，返回行列號組成的二元列表
- searchpair() 按成對關鍵字搜索
- searchpairpos() 按成對關鍵字搜索
- searchdecl() 搜索一個變量的定義

這幾個搜索函數可用於從腳本實現類似 / 的搜索命令，但有更靈活細緻的控制。先看最基本的搜索函數的參數原型 search(pattern, flag, stopline, timeout)，只有第一個參數是必須的：

• {pattern} 就是 VimL 的正則表達式，在該函數中，一些影響搜索的選項如 &ignorecase，&magic 等將影響正則表達式的解析。

• {flag} 是一個字符串，每個字符表示不同的意義，一些衝突的標誌不能並存：

◦ b 表示反向搜索，默認正向搜索；

◦ c 在光標處也能匹配成功；

◦ e 光標移動到匹配成功處的末尾，默認移動到匹配處的起始位置；

◦ n 即使匹配成功也不移動光標，但可利用函數的返回值，行地址；

◦ p 返回值不再是行地址，而是匹配成功的（或連接）子模式索引加 1；

◦ s 移動光標到匹配處前，將原位置保存在特殊標記 ‘‘ 中；

◦ w 搜索到文件末尾時，折回文件起始，與 b 並存時是到文件首折回；

◦ W 搜索到文件末尾或起始時不折回；

◦ z 從光標的列位置開始搜索，默認是從光標所在行首開始搜索。

• {stopline} 搜索從當前光標開始，可指定終止搜索的行。

• {timeout} 按毫秒數指定搜索的時間，搜索可能是個費時的操作，尤其是正則表達式寫得複雜寫得低效時，可指定時間，超時不再搜索。

所以這個函數有兩個作用，一是返回值表示匹配的行地址，另一個副作用是會移動光標，除非指定 n 標誌不移動光標。匹配失敗時返回 0 ，當然也不會移動光標。

searchpos() 函數意義一樣，只是返回多值（列表），除行號外，還返回列號，如果指定 p 標記，還返回所匹配的子模式索引（加 1）。這裏的子模式是指由或操作 \| 連接的多個模式分支，匹配其中任一個都算匹配成功，但若需要知道匹配的是哪個分支，p 標記就有用了，注意需要被索引標記的子模式還得整個放在 \(\) 中。

searchpair() 成對搜索的意義類似在 VimL 腳本（或其他類似語法的語言，需加載自帶的 matchit 插件）中在 if 關鍵字中按% 命令，它會搜索配對的 endif 以及中間的 elseif。其參數就是在 search() 的基礎上，將第一個正則表達式參數 {pattern} 換爲三個正則表達式參數 ({start}, {middle}, {end}, ...)。並且可以在可選參數 {flag} 與 {stopline} 之間再加一個可選參數 {skip} ，其意義是表示如何忽略某些匹配，比如 elseif endif 在註釋或字符串中應該是要忽略的。{skip} 是一個可執行字符串，當作表達式執行後返回非 0 就表示要忽略，執行時光標相當於已移動到匹配處。

searchpairpos() 的意義也類似，返回多值，即由行列號組成的列表而已。

searchdecl() 的作用與 gd 或 gD 普通命令類似，當然命令是取光標下的單詞，函數需要將變量名字符串當作參數傳入。

• getcharsearch() 獲得字符搜索信息

• setcharsearch() 設定字符搜索信息

這兩個函數是從 Vim8 版本新增的。字符搜索是指 f，F，t，T 這幾個命令用於實現行內搜索字符的，同時還有分號 ‘;’ 與逗號 ‘,’ 按正反向重複上次字符搜索。如果要從腳本控制這種行爲，可參考這兩個函數。

修訂窗口（quickfix）

很多命令會生成一個所謂的 quickfix 列表，這裏將其譯爲修訂。最早的應用來源是編譯源代碼給出的錯誤列表，每條項目會指出錯誤出現的文件、位置等，用於方便定位錯誤並修改。後來該概念擴展到其他許多命令，比如 grep 搜索，所以它就是一個有關定位的列表。該列表顯示在單獨的窗口中，就叫做修訂窗口，可在該窗口預覽各個“錯誤”信息，並像在普通窗口上移動，然後有方便的命令跳到相應位置外，並遍歷整個列表。

據說最早的 Vim 版本並無此功能，只是一個插件功能，後來由於功能太過強大實用，就整合爲 Vim 的內置功能了。而且還擴展出了局部修訂列表的概念，即每個窗口都可以有自己的修訂列表了。術語上，qflist 是全局的，locallist 是局部的。

- getqflist() 獲得修訂列表
- setqflist() 設置修訂列表
- getlocallist() 獲得局部修訂列表
- setlocallist() 設置局部修訂列表

getqflist() 返回的是字典列表，每個字典元素的鍵名解釋請參考相應文檔。setqflist() 接收這樣的字典列表作爲參數，並且有個可選的參數指出是添加到原修訂列表末尾還是覆蓋原列表。後兩個函數用法一樣，不過在最前面多插入一個參數指出窗口編號（不是窗口 ID）。

- taglist() 獲得匹配的 tag 列表
- tagfiles() 獲得 tag 文件列表

tag 文件是外部文件，記錄着一些 tag（如變量名、函數名、類名等需要在大項目中檢索與交叉引用的東西）的定義位置，該文件是由外部程序掃描（所有相關）源文件生成的，並遵循一定的格式。有了這樣的文件，才能使用快捷鍵 C-] 與 ‘:tag’ 命令。而 taglist() 是其函數形式，參數就是所要檢過的 tag 名稱，以正則表達式解析，要提供全名應自行加上 ^ 與 $ 界定。

Vim 使用的 tag 文件可用選項 &tags 設置，它是以逗號分隔的文件名字符串。函數 tagfiles() 返回的是當前緩衝實際所用的 tag 文件列表（VimL 列表類型）。

- complete() 設置補全列表
- complete_add() 向補全列表中增加條目
- complte_check() 檢查是否終止補全
- pumvisible() 檢查是否彈出補全窗口

插入模式下的補全是相對高級的話題。Vim 的默認模式是普通模式，定製插入模式本身就比較複雜。VimL 只提供了幾個 api 函數。complete() 是簡單地提供補全列表。complete_add() 與 complete_check() 只能用於自定義的補全函數（&compltefunc）中。

Vim 本身只是定位於通用文本編輯器，並非程序開發 IDE，但提供了這些基本接口，允許三方插件將其打造成的類似 IDE 的大多功能。尤其是 Vim8 版本新增的異步功能，能顯著增加補全的性能與可用性。此不再詳述，這些高級話題可能另闢章節討論。

命令行信息

最後看幾個有關命令行的函數。因爲命令行也是可編輯區域，也是可以通過腳本訪問的，不過一般只適於正在編輯命令行時使用，比如‘:cmap’定義的映射等。

- getcmdline() 獲得當前命令行
- getcmdpos() 獲得光標在命令行的列位置
- setcmdpos() 設置光標在命令行的列位置
- getcmdtype() 獲取命令行類型
- getcmdwintype() 獲取命令行窗口類型

命令行類型比如通過‘:’，‘/’，‘?’進入的命令行都是屬於不同的命令行類型。命令行窗口是通過特殊鍵在命令行之上再打開的一個窗口，裏面是命令行歷史記錄列表，可以方便選擇某個歷史命令或在彼基礎上作小修改後再次執行。故 getcmdwintype() 只有在命令行窗口時纔有意義，其值與 getcmdtype() 相同。

- getreg() 獲取某個寄存器的內容
- setreg() 設置某個寄存器的內容
- getregtype() 獲取某個寄存器的類型

寄存器相當於 Vim 自己管理的剪貼板，允許用戶自命名的寄存器有 26 個（即單字母表示），另外 Vim 還自動更新了許多以特殊符號表示的寄存器，各表示相應的特殊意義。寄存器的內容可用‘:registers’查看。這幾個函數則用於腳本訪問與控制寄存器。此外，對於常規字母命名的寄存器，以 @ 前綴的變量可直接表示該寄器（如 @a）。寄存器類型與選擇類型（字符、行、列塊）相同，因爲寄存器內容經常是選擇後複製進去的。

用 Vim 編輯文本要善於利用命令行與寄存，這幾個函數一般只在映射（調用）中比較有效果。

6.3 操作外部系統資源

本節介紹的函數主要着眼於訪問外部資源，比如最常用便是系統文件。

文件系統相關函數

- glob() 按文件通配符搜索文件
- globpath() 在指定目錄中搜索文件
- findfile() 在搜索路徑中查找文件
- finddir() 在搜索路徑中查找目錄

glob() 函數的作用，就相當於在 linux 終端命令 ls 所能列出的文件名。它可接收至多四個參數，只有第一個是必須的：

- {wildcard} 通配符文件名模式，非正則表達式；
- {nosuf} 讓兩個選項生效，&wildignore 可忽略某些文件，&suffixes 按文件名後綴影響結果的排序；

• {list} 提供該參數則返回列表類型，否則是用換行符分隔的字符串；

• {alllinks} 一般情況下只會找出存在的文件，對於軟鏈接文件，則其指向的文件有效才被包含在結果中，但若提供該參數，無效鏈接文件也接收。

一般第三個參數比較常用，即將結果按列表返回，以 glob(wild, 0, 1) 方式調用。

globpath() 用法是在 glob() 基礎上，額外提供一個參數指定要哪些目錄下搜索文件，必選參數，且插在第一個參數位置上。這是一個以逗號分隔的目錄名列表字符串，如 &rtp 的表示法。例如 globpath(&rtp, 'readme.md') 就能搜索出所有運行時目錄下的說明文檔（目前許多插件安裝習慣是安裝在獨立的運行時目錄下，一般會有個 readme.md 說明文檔）。

glob() 函數將返回所有匹配的文件名，但 findfile() 或 finddir() 只返回第一個匹配的文件名，一個查找文件，一個查找目錄，類似命令 ':find' 的作用。接收三個參數，只有第一個必選：

• {name} 文件名，必須是全名，不是通配符；

• {path} 在這些目錄下查找文件，也是逗號分隔的目錄列表，省略的話用選項 &path 代替。所以實際所查到的文件名類似 {first-path}/{name}。

• {count} 指定返回第幾個匹配的文件，而不是第一個，負數時返回所有匹配文件組成的列表類型變量。

Vim 的 ':find' 命令及 gf 命令使用 &path 選項值，這叫做搜索路徑，這是搜索普通文件的；不同於 &rpt 運行時路徑是搜索 vim 腳本的。搜索路徑同時支持向下搜索與向上搜索的機制，在 {path} 參數或 &path 選項中使用特殊字符達成：

• 向下搜索：* 表示任意字符，** 表示任意子目錄；

• 向上搜索：{one-path};{upto-path},{another-path} 即在一個路徑（逗號分隔的）末尾再加一個分號，接一個相對基目錄（{one-path}）更上層的目錄（{upto-path}）就能從指定目錄開始向上搜索，依次在其父目錄搜索，直到終止目錄 {upto-path}。終止目錄可省，但分號不可省，否則在該目錄中就認爲不需要支持向上搜索。建議不限定終止目錄時寫成 {base-path};/, 或 {base-path};~, 一直上溯到系統根目錄或自己的家目錄。

• 相對 Vim 當前路徑寫成單點 ':'，相對當前正編輯的文件緩衝的路徑寫成 ':/'。

向上搜索機制，對於搜索工程項目文件很有用。比如當你正在編輯一個源代碼文件，它一般被組織在各層子目錄下，要找到項目文件就得使用向上機制了，例如.git/ 目錄或 tags 文件，都一般放在項目頂層目錄中。

• resolve() 解析鏈接文件名

• simplify() 簡化文件名路徑

• pathshorten() 縮寫文件名的中間路徑

• fnamemodify() 文件名修飾

resolve() 是處理軟鏈接文件（linux 系統）或快捷方式（MS-Windows 的.lnk）的，將其轉爲實際指向的文件名。在其他系統同 simplify() 簡化處理。文件名需要簡化的一個例子是包含一系列的點號與雙點號，如./dir/.././/file/ ，這可能是由其他函數拼接而來。simplify() 簡化後不改變其意義，如上例簡化結果爲./file/。但是 pathshorten() 只是簡單地將中間路徑都縮寫至首字母，顯然是不保證其有效意義的。比如在默認的多標籤頁的名字，爲節省屏幕空

間就將當前編輯文件縮寫目錄名，~.vim/autoload/myfile.vim 將簡寫爲 ~/.v/a/myfile.vim（如果覺得這比較醜，可尋插件定製標籤頁欄）。

文件名修飾是指如何從一個文件名中獲取其目錄、全路徑名、後綴名等相關的名字字符串。函數 fnamemodify({fname}, {mods}) 的第二參數就叫做修飾符，修飾符以冒號開頭帶一個單字母表示不同意義，且可連續使用。主要的修飾符如：

- ':p' 文件全路徑名
- ':h' 父目錄名（文件名頭部，去除路徑分隔符最後一部分）
- ':t' 文件名尾部（一般是 ':h' 剩余部分，純文件名）
- ':e' 文件名後綴
- ':r' 文件名主體（相對於 ':e' 而言，不包括後綴，但可能包含父目錄）

注意 fnamemodify() 不處理特殊文件名變量，需用 expand() 先展開，不過後者也可以直接加修飾符後綴。如以下兩個語句等效：

```
: echo fnamemodify(expand('%'), ':p:t')
: echo expand('%:p:t')
```

- executable() 檢查是否可執行程序
- exepath() 可執行程序的全路徑
- filereadable() 文件是否可讀
- filewriteable() 文件是否可寫
- getfperm() 獲取文件權限（類 rwxrwxrwx 字符串）
- getftype() 獲取文件類型
- isdirectory() 檢測目錄是否存在
- getfsize() 獲取文件字節大小（目錄返回 0）
- getftime() 獲取文件的最後修改時間（整數，按秒計）

這幾個函數用於檢查指定文件的屬性，其中 getftype() 返回的字符串主要有如：

- file 普通文件
- dir 目錄
- link 軟鏈接文件
- bdev，cdev，socket，fifo，other 等
- getcwd() 獲取當前工作路徑
- haslocaldir() 檢測當前窗口是否有局部當前路徑（:lcd）

這兩個函數都可選帶兩個參數，指定窗口編號與標籤頁編號，因爲取當前窗口的當前路徑。Vim 啓動時，從 shell 環境中繼續當前路徑，這是全局當前路徑，可用 ':cd' 命令修改。每個窗口可有自己的局部當前路徑，這用 ':lcd' 修改。如果從未用過':lcd' ，窗口的局部當前路徑就與全局當前路徑相同。新分裂的窗口繼承原窗口的當前路徑。':pwd' 打印的是全局當前路徑，因此有可能與 getcwd() 不同。

- mkdir() 創建新目錄（類似 $mkdir）
- delete() 刪除文件（類似 $rm）

- rename() 重命名文件（類似 $mv）
- readfile() 讀文件至一個字符串列表
- writefile() 將字符串列表寫入文件

這幾個文件操作函數，除了 readfile() 返回列表外，其他函數在操作成功時返回 0，失敗時返回非零錯誤碼。其功能與相應的 linux 命令類似，不過將命令行參數改成函數調用參數。如 mkdir('name', 'p') 類似 shell 命令 $mkdir -p name 可以自動創建中間目錄；delete() 刪除非空目錄時必須加參數 rf（謹慎）；rename() 重命名文件可能覆蓋已有文件無警告。當然這些操作也涉及系統權限。

讀文件函數支持三個參數，readfile(fname, binary, max) ，後兩個是可選的。默認是按文本格式讀入，主要會處理換行符。如果提供 {binary} 參數，按二進制格式讀入，雖然也會根據換行符分隔爲列表元素，但元素中可能再保留回車符（dos 格式的文件），且最後可能多加一個空元素（若文件末尾是換行符）。第三個可選參數 {max} 可指定只讀入前幾行，類似 linux 命令 $head -n ，但如果 {max} 參數是負數，則只讀入末尾幾行，類似命令 $tail -n。

寫文件函數要求兩個參數，作爲內容的字符串列表，以及文件名，還有個可選參數標記：writefile(list, fname, flags)。標記 {flags} 若包含 b 則按二進制格式寫入，若包含 a 則添加到原文件末尾，否則是覆蓋原文件。

一般情況下，Vim 是處理文件文本的，在使用這兩個讀寫文件函數時，沒必要指定 b 二進制格式。但是按二進制格式先 readfile() 再 writefile() 確實能達到複製文件的作用。

調用外部系統命令

在 Vim 的命令行中，可用 ':!' 歎號開頭，調用外部系統命令。而在 VimL 脚本中，相應功能的函數是 system()。

- system() 執行系統命令，結果爲字符串形式返回
- systemlist() 執行系統命令，結果以列表形式返回
- libcall() 調用外部庫函數，結果返回字符串
- libcallnr() 調用外部庫函數，結果返回數字

system(cmd, input) 將字符串 {cmd} 當作系統命令執行，返回字符串結果。如果 {cmd} 命令需要輸入，則可提供可選參數 {input} ，一般也是字符串，首先寫入臨時文件，再當作標准輸入傳給 {cmd} 。如果 {input} 是字符串列表，則以二進制 b 方式調用 writefile() 寫入臨時文件。

{cmd} 命令字符串不支持管道。並且爲了安全與正確性起見，最好調用 shellescape() 轉義特殊字符。systemlist() 用法類似，只是返回結果是字符串列表。

libcall() 類似於 call() 的基礎用法，只是調用外庫（'.so' 或 '.dll'）的函數，故需要庫名、函數名與參數列表。當然不能隨意調用外部庫，只能調用專爲擴展 vim 的庫，那才比較安全與實用。該函數將結果返回爲字符串，另一個 libcallnr() 函數返回的是數字結果。

- hostname() 獲取 vim 所在運行的系統（計算機）名字
- getpid() 獲取 vim 運行的進程號 PID

• tempname() 獲取可用於臨時文件的文件名

在實現比較複雜的功能時，可能需要用到臨時文件，用 tempname() 獲得一個可用的文件名（保證不重名）。也可以自己根據進程 PID 構建有規律的臨時文件名。

日期時間函數

• localtime() 獲取當前時間
• strftime() 格式化時間
• reltime() 獲取相對時間
• reltimestr() 將相對時間轉爲字符串
• reltimefloat() 格式化相對時間轉爲浮點數

localtime() 用於獲取當前的標准時間，即從 1970 年至今的秒數。將這樣的時間轉爲可讀模式，用 strftime(format, time) 函數，缺省 {time} 參數時，取當前時間，相當於先調用 localtime()。可用時間格式 {format} 與 C 語言的同名標准函數相同，如 strftime('%Y-%m-%d') 將返回類似 2017-11-11 的字符串。

reltime() 返回更精確的時間，具體格式與系統有關。無參數調用返回當前時間，一個參數 reltime(start) 返回從開始時刻（{start} 也應該是由該函數返回的）到現在所經過的時間，兩個參數 reltime(start, end) 返回兩個時刻之間的時間。用 reltimestr() 將這樣的時間轉爲字符串表示，reltimefloat() 轉爲浮點數表示，因爲字符串表示法也正像個浮點數（即秒數加小數點加毫秒數）。因其精確到毫秒，可用來計算命令或函數執行的時間。

用戶交互函數

• input() 獲得用戶從命令行輸入的一行文本
• inputsave() 保存用戶輸入序列
• inputrestore() 恢復用戶輸入序列
• inputsecret() 按密文輸入
• intputdialog() 從對話框中輸入一行文本

在 VimL 脚本中與用戶交互的最常用的函數是 input(提示, 默認值, 補全方法)。提示字符串參數必須給，可以是空字符串，也可以用 \n 表示多行提示。後面兩個參數可選。Vim 首先在命令打印提示字符串，等待用戶輸入一行文本，按回車返回用戶剛輸入的這行文本。如果直接回車沒任何輸入，則返回傳給函數的默認值（或空字符串）。當用戶輸入時，相當於編輯命令行，所以爲便於用戶輸入，可提供補全方法，類似自定義命令那般。而且用戶的輸入也有獨立的命令行歷史記錄。

顯然，input() 函數不宜用於啓動配置 vimrc 中。此外，也要避免用於映射中，因爲映射的後續鍵相當於用戶輸入，會當作 input() 的迴應輸入。如果一定要用於映射中，請在調用 input() 前後分別調用 inputsave() 與 inputrestore()。

inputsecret() 用法一樣，只是用戶輸入的文本不直接顯示在屏幕命令行中，以星號 * 代替。此外也不支持補全，不放入歷史記錄中，因爲這主要用於提示輸入密碼。

在 GUI 版本中,inputdialog() 可彈出對話框,讓用戶從對話框中輸入,否則類似 input() 函數。

• inputlist() 讓用戶從一個列表中選擇一項

• confirm() 也是讓用戶從列表中選擇一項

inputlist() 接收一個字符串列表參數,Vim 將每個元素一行顯示在命令行上方的消息區,然後提示用戶輸入一個數字選擇一項(GUI 版本可用鼠標)。注意按列表索引慣例,0 表示選擇第一項。爲彌補這個反人類設計,這有個技巧: 將提示文本寫在列表的第一項,後續有效選項字符串也以索引 '1.', '2.' 之類的開始,讓用戶能直觀地選擇數字。

讓用戶做選擇還有另一個函數 confirm() ,它可用於 GUI 版本,也可用於終端版本。它可接收四個參數,confirm(提示,選項列表,默認選項,對話框類型),一般只用到前兩個。與 inputlist() 不同的是,提示文本爲獨立參數,且選項列表是字符串,用回車分隔每一選項,且第一項是 1 。在每一項的字符串中,可以將 & 加在某個字符之前,則按該字符時直接選擇了項目(選擇快捷鍵),且不像 inputlist() 那樣會將所按鍵顯示在命令行中(因爲其實這是爲 GUI 版本設計的),也不需要多按回車確認,就是快捷鍵直接選擇。當然函數返回的仍是選項索引,並非快捷鍵字符。可選的默認選項參數也應該是數字索引,不提供時默認 0 ,算是無效選項。

• getchar() 獲取用戶按下的下一個鍵

• getcharmod() 獲取用戶按鍵時的修飾鍵

• feedkeys() 將一個字符串放入 vim 待響應的按序序列

getchar() 用於獲取用戶(或輸入流)的下一個鍵。不同於 input() 進入命令行等待交互,而是默默地等待獲取下一輸入鍵。相對細節很多,用到時請參考文檔。因爲 vim 本身的總體工作(消息)循環,就是等待用戶按鍵,然後作出不同響應。

getcharmod() 用於獲取修飾鍵(收到上個鍵時同時按下的修飾鍵),如 shift = 2、ctrol = 4 、alt = 8 等。將可用修改鍵用二進制編碼,返回一個數字就能表示哪些修飾鍵被按下了。

feedkeys() 的用途就比較詭祕了。它把一個字符串放回輸入流中,當作是用戶的按鍵輸入序列。特殊按鍵用"
< 標記 >" 表示。默認情況下,放回的這些字符鍵是可再被重映射的,然而也有一些可選參數控制細節。

• browse() 打開瀏覽文件對話框

• browsedir() 打開目錄選擇對話框

這兩個函數只能用於 GUI 版本,彈出標准對話框,讓你選擇一個文件或目錄,返回所選擇的文件路徑名。可以傳入參數指定對話框標題及初始瀏覽目錄。

• getfontname() 獲取當前所用的字體

• getwinposx() 獲取 gVim 窗口的座標

• getwinposy() 獲取 gVim 窗口的座標

這幾個函數只能用於 GUI 版本,檢索 GUI 才用得到的信息。

異步通訊函數

自 Vim8 版本引入了一些全新的特性：任務（job）、定時器（timer）、通道（channel），這都涉及異步編程，主要通過回調函數實現功能。爲此也提供了一系列相關的內建 api 函數。不過本章不想羅列這些函數，畢竟需要理解相應的功能纔有理解函數用法的意義。留待後續章節專門討論吧。

6.4 其他實用函數

在本章的最末，再介紹一些不太分類，或不常用（但不甚複雜）的函數。須要再次強調的是，VimL 的函數，是爲了訪問或（與）控制相應的 Vim 功能而設的。必須理解相應的功能才能用好相應的函數。Vim 提供的功能很多，對於個體用戶而言，可能對某些功能並不關注或並不感興趣。

因此，本章羅列介紹這些函數，無法求細緻，只爲說明 VimL 有這麼一種類的函數可用。當你真正需要用到時，再回去查手冊。任一編程語言的 api 函數，只能通過手冊學習，通過實踐提高。而這無法從任一書籍教程中學得，書籍只能是引入門，幫用戶建立個相關概念而已。

特性檢測函數拾遺

- has() 當前 Vim 版本是否編譯進某個功能，似 ':verstion' 功能
- exists() 檢查是否存在某個變量、命令、函數等對象
- type() 返回變量類型

其中 exists() 函數功能強大，用參數字符串前綴來表示哪類對象，主要有以下幾種：

- 選項：&option_name 只判斷選項存在，+option_name 判斷選項是否有效
- 環境變量：$ENV_NAME
- 函數：*function_name
- 命令：:command_name 對於函數與命令，都可檢查內建的或自定義的
- 變量：variable_name 即沒有特殊前綴時檢查變量是否存在
- 自動事件：#event，#group 等
- hasmapto() 檢查某個命令（鍵序列）是否有映射（{rhs}）
- mapcheck() 檢查某個按鍵序列是否有被映射（{lhs}）
- maparg() 獲得某個被映射的鍵序列實際映射鍵序
- wildmenumod() 在命令行映射中檢查是否出現補全模式

這幾個檢查映射狀態的函數，常用於設計可定製插件的映射，避免重複、覆蓋或冗余的映射。maparg() 與 mapcheck() 的主參數都是映射的 {lhs}，前者要求精確匹配，後者只需匹配前綴，返回映射到的 {rhs}。hasmapto() 是反向檢查是否有任意的 {lhs} 映射到參數指定的 {rhs}。

類型文件語法相關

文件類型是 Vim 的重要概念，簡單理解的話，不同的文件名後綴往往代表不同類型的文件（&filetype）。比如不同編程語言的源代碼文件。不過 Vim 只是編輯器，它並不能理解（編譯或解釋）任一種編程語言（VimL 除外）。所以 Vim 語境下的 “語法”，本質上只是 “詞法”，旨在說明可以如果對文件的不同部分進行不同的高亮着色，不過習慣上仍稱爲語法着色。

這主要分兩步實現。首先是定義高亮組（highlight group），它說明 “如何高亮”，描敘了該高亮下的顏色、字體等信息。然後是定義語法項（syntax），它說明高亮什麼，主要是基於正則表達式，將不同匹配（match）部分應用不同的高亮組。這樣就有可能將一個緩衝文件在窗口中以五顏六色的方式呈現出來。

一般不同的文件類型有相應的語法文件，文件中主要用 ‘:highlight’ 與 ‘:syntax’ 命令定義了各種高亮組與語法項。這裏要介紹的 VimL 內置函數，可以檢索當前緩衝中實際生效的語法信息，以及臨時增加修改部分高亮方式。

- matchadd() 增加一種匹配應用某種高亮組
- matchaddpos() 在特定（行列）位置上匹配應用高亮
- matchdelete() 刪除一處匹配的高亮
- clearmatches() 清除所有匹配高亮
- matcharg() 獲取 ‘:match’ 命令的參數信息
- getmatches() 獲取所有自定義匹配高亮
- setmatches() 恢復一組自定義匹配高亮

這幾個函數用於手動控制一些文本的高亮方式（相對於語法文件的自動加載）。雖然這種方式似乎比較原始，但有助於理解 Vim 語法高亮的處理機制，並且在臨時微調語法高亮或處理一些簡單非常規文件類型時也有奇效。

首先要了解 ‘:match’ 命令，它與之前介紹的用於匹配字符串的 match() 函數是不同的功能。‘:match{group} /{patter}/’ 的意思相當於臨時定義一種語法匹配，將匹配正則表達式的文本應用指定的高亮組。你可以初步地認爲每種文件類型的特定語法文件中都正式地定義了很多種類似的語法匹配。不過 ‘:match’ 的臨時定義只能定義三種不同的匹配，另外兩個命令是 ‘:2match’ 與 ‘:3match’。實際上它只是 ‘:match’ 命令的數字參數，默認是 ‘:1match’ 而已。限制三種可能是出於性能、管理與實用的綜合考慮。

然後 matchadd() 是 ‘:match’ 命令的函數方式。所不同的是它不限於只定義三種匹配，可以調用這個函數定義許多匹配，並且返回不同的 ID 用以表示所定義的不同匹配。當然 1-3 這前三個 ID 被保留給 ‘:match’ 命令使用。

matchaddpos() 的功能類似，不過它不是通過正則表式式來定義匹配，而是准確地給出一組位置信息，說明在哪些行（列）上要高亮。因爲基於正則的語法高亮是比較低效的，（在一些舊機器上打開大文件時若發現卡，可嘗試關閉語法着色），按位置高亮就不那麼耗性能了。

matchdelete() 用於刪除自定義匹配，參數就是 matchadd() 返回的匹配 ID（或者 1-3 代表通過 ‘:match’ 命令定義的）。clearmatches() 則清除所有自定義匹配，不需要參數。

matcharg() 就是用於查看之前的自定義匹配，接收匹配 ID 爲參數，返回其定義的信息。getmatches() 用於獲得所有自定義匹配的詳細信息，返回的是字典列表，它可傳給 setmatches() 恢復自定義匹配。

- hlexists() 由高亮組名判斷其是否存在
- hlID() 由高亮組名返回其數字 ID
- synID() 返回當前緩衝指定行列位置處所使用的語法項 ID
- synIDattr() 由語法項 ID 返回可讀的屬性信息
- synIDtrans() 返回一個語法項 ID 實際所鏈接的語法項 ID
- synstack() 返回當前緩衝指定位置處所有的語法項 ID 堆棧
- synconcealed() 返回當前緩衝指定位置處的隱藏語法信息

在 Vim 內部，爲每個高亮組與語法項都分配了 ID，其中高亮組有名字，而語法項是個更虛擬的概念，沒有名字，只能用 ID 表示。在當前緩衝的某部分文本（以行列號爲參數）使用了什麼語法高亮，可用 synID() 獲得，據此可進一步由 synIDattr() 獲得該語法高亮的詳情屬性。

在語法文件中，普遍使用 ':highlight link' 命令鏈接高亮組。因爲 Vim 預設了大量通用的高亮組名字，但允許用戶在爲不同類型文件的語法中使用更有意義的高亮組名，爲避免重複定義高亮屬性，就可以將新高亮組鏈接到既有高亮組。synIDtrans() 函數就是用於獲得某個語法項 ID 實際鏈接的語法項 ID。

Vim 的語法定義其實不是簡單的正則匹配，還有更復雜的區域（region）與嵌套規則。於是對於一部分文本（緩衝的行列位置）而言，它可能不只應用了一個語法項高亮，而是由內向外形成了語法高亮棧。synstack() 就是獲取這樣的棧的函數，它返回一個列表，最後一個元素其實就是 synID()。

- foldclosed() 檢查當前緩衝指定行是否被摺疊，返回摺疊區的第一行
- foldclosedend() 同上，返回摺疊區的最後一行
- foldlevel() 返回當前緩衝指定行應該摺疊的最深層次，不要求已摺疊
- foldtext() 默認的摺疊行顯示文本計算函數
- foldtextresult() 當前緩衝指定行如果摺疊，摺疊行應該顯示的最終文本

摺疊從某種意義來說，也屬於語法規則的範疇，只是它不是關於如何着色的，而是定義文本層次的。摺疊方法有很多種，可由選項 &foldmethod 指定，其中一種就是 syntax 由語法定義決定摺疊層次。有很多普通命令（z 開頭的系列）處理摺疊。這裏的幾個函數只是訪問摺疊信息。

如果當前緩衝的某行已被摺疊中，函數 foldcolsed() 與 foldclosedend() 分別返回整個摺疊區的首行與尾行；若未摺疊，返回 '-1'。據此可知緩衝的實際文本與窗口顯示的文本行的差異。foldlevel() 返回摺疊層級。任何摺疊方法最終都由每行的摺疊層級決定摺疊行爲。其中有種自定義摺疊函數（表達式），就由用戶自己控制、計算返回每行的摺疊層級。其他摺疊方法 Vim 會自動計算摺疊層級。

摺疊後，會在摺疊首行顯示一行特徵文本，這行文本的計算方法由選項 &foldtext 配置指定。其默認值就是 foldtext()，該函數也只能在計算 &foldtext 時調用。某行摺疊後實

際將顯示的特徵文本，則由 foldtextresult() 給出。

測試函數

當程序或腳本變得複雜起來，單元測試就可能很有必要了。從 Vim8 版本始，也提供了許多函數方便用戶寫單元測試。

- assert_equal() 斷言兩個值相等，包括變量類型相同
- assert_notequal() 斷言不相等
- assert_inrange() 斷言一個值處在某個範圍
- assert_match() 斷言匹配正則表達式
- assert_notmatch() 斷言不匹配正則表達式
- assert_false() 斷言邏輯假，或數字 0
- assert_true() 斷言邏輯真，或非 0 數字
- assert_exception() 斷言會拋出異常
- assert_fails() 斷言執行一個命令會失敗

這些斷言函數用法類似，一般是接收預期值與實際值，如果不滿足斷言條件，就添加一條消息至內置變量 v:errors 列表中，其中消息字符串能傳入可選參數定製。這些函數本身不會產生錯誤輸出或中斷運行，只是先將錯誤信息暫存至 v:errors 中。所以在做單元測試中，應該先將 v:errors 列表置空，調用一系列斷言函數，最後檢查該列表保存了哪些錯誤消息，如果一切正常，該列表應該是空的。

第七章　VimL 面向對象編程

面向對象是一種編程思想，並不特指某種編程語言。所以不必驚訝用 VimL 也能以面向對象的方式來寫代碼。本章先簡單介紹一下面向對象的編程思想，再探討如何利用 VimL 現有的特性實現面向對象編程。最後應由用戶自行決定是否有必要使用面向對象的風格來寫 VimL 腳本。

7.1　面向對象的簡介

在前文中用了比較多的篇幅來介紹函數。如果主要以函數作爲基本單元來組織程序（腳本）代碼，函數間的相互調用通過參數傳遞數據，這種方式或可稱之爲面向過程的編程。大部分簡單的 VimL 腳本都可以通過這種方式實現。單元函數的定義與複用也算簡潔。

但是，如果有更大的野心，想用 VimL 實現較爲複雜的功能時，只採用以上基於函數的面向過程編程方式，可能會遇到一些鬧心的事情。比如函數參數過多，需要特別小心各參數的意義與次序，或許還可能不可避免要定義相當多的全局變量。當然，這可能並不至於影響程序的功能實現，主要還是對程序員維護代碼造成困擾，增加程序維護與複用的困難。

這時，就可考慮面向對象的編程方式。其核心思想是數據與函數的整合與統一，以更接近人的思維方式去寫代碼與管理代碼。

面向對象的基本特徵

按一些資料的說法，面向對象包含以下四個基本特徵：

- 抽象
- 封裝
- 繼承
- 多態

嚴格說來，任何編程都應該從抽象開始。分析現實需求問題的主要關係，歸納出功能單元組成部分。按面向過程編程方式設計函數時，同樣也要求程序員的抽象能力。所以也有些教程資料說面向對象的基本特徵是後面這三個：封裝、繼承與多態。這又涉及面向對象的另一個關鍵概念，類。

類就是將現實諸問題抽象後的封裝結果。它包括數據以及操作這些數據的方法，從概念及表現上將這兩部分放在一起視爲一個整體，就稱之爲封裝。類往往對應着現實世界的某種類型的實體或動作。我們一般只用關注某類事物的表面接口，而不必關心其內部構造細節。反映到程序上，類的封裝就是爲了隱藏實現，簡化用法，一般用戶只要理解某個類是什麼或像什麼，以及能做什麼，則不用深究怎麼做。

所以在程序中，類不外是一種可自定義的複雜類型。與之相對應的簡單類型就是如數字、字符串這種在大多數語言都內置支持的。簡單類型除了可以用值表示一種意義外，還支持特定的操作，如數字的加減乘除，字符串的聯連、分割、匹配等。類也一樣，它用於表示值的就是被封裝（可能多個）數據，也常被稱爲成員屬性，它所支持的操作方法也叫成員函數。而對象與類的關係，也正如變量與類型的關係。對象是屬於某個類的，有時稱其爲實例變量。

有些語言的面向對象還對類的封裝進行了嚴格的控制，比如從外部訪問對象只能通過類提供的所謂公有方法（屬性），而另外一些私有方法（屬性）只能在類內部的實現中使用。

繼承是爲了拓展封裝之後的類代碼的複用，將一個類的功能當作一個整體複用到另一個相關的類中。這也是對現實世界中具有某種從屬關係的事件的一種抽象。在被繼承與繼承的兩端，一般稱之爲基類與派生類，或通俗點叫父類與子類。子類繼承了父類的大部分屬性與方法（具體的語言或由於訪問權限另有細節控制），因而可以像操作父類一樣操作子類。

多態是爲了進一步完善繼承的用途而伴生的一個功能實現概念，使得在一簇繼承體系中，諸派生類各具個性的同時，也保留共性的方法訪問接口。即向許多對象發送相同的消息使其執行某個操作，各對象能依據其類型作不同的響應（功能實現）。

面向對象示例分析

先舉個概念上的例子。就比如數字，僅管很多語言把數字當作簡單的內置類型來處理，卻也不妨用類與對象的角度來思考這個已經被數學抽象過的概念。

我們知道數字有很多種：整數、實數、有理數、複數等。每種數都可以抽象爲一個類，還可以在這之上再抽象出一種虛擬的 “數字” 類，當作這些數類的統一基類。這些類簇之間就形成了一個繼承體系。凡是數字都有一些通用方法，比如說加法操作。用戶使用時，只需對一個數字調用加法操作，而不必關心其是哪類數，每類數會按它自己的方式相加（如有理數的相加與複數的相加就有顯著不同）。這就是使用上的多態意義。整數一般可以用少數幾個字節（四或八字節）來表示，但如果有時要用到非常大的整數，可能需要單獨再定義一個無限制的大整數類。但對一般用戶來說，也不必關心大整數在底層如何表示，只需按普通小整數一樣使用即可，這就是封裝的便利性。

再舉個切近 Vim 主題的例子。Vim 是文本編輯器，它主要處理的業務就是純文本文件。那麼就不妨將文本文件抽象爲一個類。其實從操作系統的角度講，文件包括文本文件與二進制文件，若按 “一切皆文件” 的 linux 思想，其他許多設備也算文件。然而，以 Vim 的功能目的而言，可不必關心這些可擴大化的概念，就從它能處理的文本文件作爲抽象的開始吧。

Vim 將它能編輯的文件分爲許多文件類型，典型的就如各種編程語言的源代碼文件。於是每種文件類型都可視爲文本文件這個 “基類” 的 “派生類”。然後，Vim 所關注的只是編輯源碼，並不能編譯源碼，它只能處理表面上的語法（或文法）用於着色、縮進、摺疊等美化顯示或格式化的工作。所以不妨再把一些 “語法類似” 的語言再歸爲同一類，比如 C/C++、java、C#、javascript 等（都以大括號作爲層次定界符），就可以在其上再抽象一個 C-Family 的基類，它處於最基本的文本文件之下，而在各具體的文件類型之上。顯然，類的抽象與設計，是與特定的功能目標有關的。若在其他場合，將 C++、java、javascript 等傻傻分不清混爲一談就可能不適合了。

若再繼續分析，C 語言與 C++ 語言還算是不同的語言，總有些細節差異需要注意，尤其是人爲規定的源碼編程風格問題。至於是否真要再細分爲兩個類，那得看需求的權衡了。另外，C/C++ 語言還有個特別的東西，它要分爲頭文件與實現文件。這也得看需求是否要再劃分爲兩個類設計。如果在編寫 C/C++ 代碼時需要經常在頭文件聲明與實現文件的實現時來回跳轉，甚至想保持實現文件的順序與頭文件聲明一致以便於對照閱讀，那麼再繼承兩個類分別設計或許是有意義的呢。

對所有這些語言源碼文件，Vim 都提供了一個縮進重格式化的功能（即 = 命令）。只要爲每個類實現重縮進的操作（實際上利用了繼承後，也只要在那些有差異需求文件類型上額外處理），就可以讓 Vim 用統一的一鍵命令完成這個工作了。這就相當於多態帶來的便利。

當然了，以上的舉例，只是概念上的虛擬示例。Vim 編程器本身是用 C 語言寫的，並沒有用到面向對象的方式，因而也不會爲文件類型設計什麼類。而且既然它主要是爲處理文本，VimL 也只要處理簡單的整數與實數（浮點數）即可，不會去設計其他複雜的數字類。這主要是說明如何採用面向對象的思想分析問題，提供一種思路與角度，順便結合示例再說明下面向對象的幾個特徵。

面向對象的優劣提示

上文介紹了面向對象的特徵，由此帶來代碼易維護易管理的優點。同時上面的例子也說明面向對象並不是必要，不用面向對象也能做出很好的應用產品。

其實，面向對象主要不是針對程序，而是針對程序員而言的。如果簡單功能，單人維護，尤其是一次性功能，基本就不必涉及面向對象，因爲要實現對象的封裝會增加許多複雜代碼。面向對象適合的是複雜需求，尤其涉及多人協作或需要長期維護的項目。此外，在實際使用面向對象編程時，也要注意避免類的過度設計，增加不必要複雜度。

本章剩下的內容旨在探討如何使用 VimL 實現基本的面向對象。從學習的角度而言，也可據此更深入地瞭解 VimL 的語言特性。至於在實踐中，開發什麼樣的 Vim 功能插件值得使用面向對象編程，那就看個人的需求分析與習慣喜好了。

7.2 字典即對象

字典是 VimL 中最複雜全能的數據結構，基於字典，幾乎就能實現面向對象風格的編程。在本章中，我們提到 VimL 中的一個對象時，其實就是指一個字典結構變量。

按對象屬性方式訪問字典鍵

首先要了解的是一個語法糖。一般來說，訪問字典某一元素的索引方法與列表是類似的，用中括號 [] 表示。只不過列表是用整數索引，字典是用字符串（稱爲字典的鍵）索引。例如：

```
: echo aList[0]
: echo aDict['str']
```

（這裏假設 aList 與 aDict 分別是已經定義的列表與字典變量）

如果字典的鍵是常量字符串，則在中括號中還得加引號，這寫起來略麻煩。所以如果字典鍵是簡單字符串，則可以不用中括號與引號，而只用一個點號代替。例如：

```
: echo aDict.str
```

這就很像常規面🄴對象語言中訪問對象屬性的語法了。所謂簡單字符串，就是指可作爲標識符的字符串（比如變量名）。例如：

```
: let aDict = {}
: let aDict['*any_key*'] = 1
: let aDict._plain_key_ = 1
: let aDict.*any_key* = 0      |" 肯定語法錯誤
```

然後要提醒的是，字典鍵索引也可用字符串變量，那就不能在中括號內用引號了。當要索引的鍵是變量時，中括號索引語法是正統，用點號索引則不能達到類似效果，因爲點號索引只是常量鍵的語法糖。例如：

```
: let str_var = 'some_key'
: let aDict[str_var] = 'some value'
: echo aDict[str_var]    |" --> some value
: echo aDict.some_key    |" --> some value
: echo aDict.str_var     |" 未定義鍵，相當於 aDict['str_var']

: echo aDict.{str_var}    |" 語法錯誤
: let prefix = 'some_'
: echo aDict.{prefix}key    |" 語法錯誤
: let prefix = 'a'
: echo {prefix}Dict.some_key |" --> some value
```

```
: let midfix = '_'
: echo aDict.some{midfix}key   |" 語法錯誤
```

上例的後半部分還演示了 VimL 的另一個比較隱晦（但或許有時算靈活有用）的語法，就是可以用大括號 {} 括住字符串變量內插拼接變量名，這可以達到使用動態變量名的效果。但是，這種拼接語法也只能用於普通變量名，並不能用於字典的鍵名。鍵名畢竟與變量名不是同種概念。（關於大括號內插變量名的語法，請參閱 ':h curly-braces-names'）

總之，將字典當作對象來使用時，建議先創建一個空字典，再用點索引語法逐個添加屬性。可以在用到時動態添加屬性，不過從設計清晰的角度看，儘可能集中地在一開始初始化主要的屬性，通過賦初值，還可揭示各屬性應保存的值類型。例如，我們創建如下一個對象：

```
: let object = {}
: let object.name = 'bob'
: let object.desc = 'a sample object'
: let object.value = 0
: let object.data = {}
```

從上例中便可望文生義，知道 object 有幾個屬性，其中 name 與 desc 是字符串類型，value 是一個數字，可能用於保存一個特徵值，其他一些複雜數據就暫存 data 屬性中吧，這是另一個字典，或也可稱之爲成員對象。當然了，VimL 是動態類型的語言，在運行中可以改變保存在這些屬性中的值的類型，然而爲了對程序員友好，避免這樣做。

字典鍵中保存對象方法

如果在字典中只保存數據，那並不是很有趣。關鍵是在字典中也能保存函數，實際保存的是函數引用，因爲函數引用纔是變量，才能保存在字典的鍵中，但在用戶層面，函數引用與函數的使用方式幾乎一樣。

保存在同一個字典內的數據與函數也不是孤立的，而應是有所聯繫。在函數內可以使用在同一個字典中的數據。用形象的話，就是保存在字典內的函數可以操作字典本身。這就是面向對象的封裝特性。字典鍵中的函數引用，就是該對象的方法。

在 VimL 中，定義對象的方法也有專門的語法（糖），例如：

```
function! object.Hello() abort " dict
    echo 'Hello ' . self.name
endfunction

: call object.Hello()    |" --> Hello bob
```

在上例中，object 就是已經定義的字典對象，這段代碼爲該對象定義了一個名爲 Hello 的方法，也即屬性鍵，保存的是一個匿名函數的引用；在該方法函數體內，關鍵字 self 代

表着調用時（不是定義時）的對象本身。然後就可以直接調用‘:call object.Hello()’了，在執行該調用語句時，self 就是 object。

按這種語法定義對象方法時，可以像定義其他函數一樣附加函數屬性，其中 dict 屬性是可選的，即使不指定該屬性，也隱含了該屬性。之所以說這也像是一個“語法糖”，是因爲這個示例相當於以下寫法：

```
function! DF_object_Hello() abort dict
    echo 'Hello' . self.name
endfunction
let object.Hello = function('DF_object_Hello')
```

這裏函數定義的 dict 屬性不能省略，否則在函數體內不能用 self。不過這仍是僞語法糖，因爲這兩者並不完全等效，後者還新增了一個全局函數，污染了函數命名空間。而上節介紹的點索引屬性，object.name 纔是與 object[‘name’] 完全等效的真語法糖。

從 VimL 的語法上講，在字典鍵中保存的函數引用，可以是相關的或無關的函數。但從面向對象設計的角度看，若往對象中添加並無關聯的函數，就很匪夷所思了。例如下面這個方法：

```
function! object.HellowWorld() abort dict
    echo 'Hello World'
endfunction
```

在這個方法內並未用到 self，也就是說不需要對象數據的支持，那強行放在對象中就很沒必要了。要實現這個功能，直接定義一個名爲 HelloWorld 的全局函數（或腳本局部函數）就可以了。

然而，一個函數方法是否與對象有關，這是一種抽象分析的判斷，並非是從語法上函數體內有無用到 self 來判斷。假如上面這個 object 的用於打招呼的 Hello() 方法另增需求，除了打印自身的名字外，還想附加一些語氣符號。我們也將這個附加的需求抽象爲函數，修改如下：

```
function! object.EndHi() abort dict
    return '!!!'
endfunction

function! object.Hello() abort dict
    echo 'Hello ' . self.name . self.EndHi()
endfunction

: call object.Hello()    |" --> Hello bob!!!
```

這裏的 EndHi() 方法也不需要用到 self，不過從它的意途上似乎與對象相關，所以也

存在字典對象的鍵中，也未嘗不可。

在一些面向對象的語言中（如 C++），這種用不到對象數據的方法可設計爲靜態方法，它是屬於類的方法，而不是對象的方法。那麼，在 VimL 中，可以如何理解類與對象的區別呢?

複製類字典爲對象實例

事實上，VimL 只提供了字典這種數據結構，並沒有什麼特殊的對象。所以類與對象都只能由字典來表示，從語法上無從分辨類與對象，這隻能是由人（程序員）來管理。通過某種設計約定把某個字典當作類使用，而把另一些字典當作對象來使用。

這首先是要理解類與對象的關係。類就是一種類型，描敘某類事物所具有的數據屬性與操作方法。對象是某類事物的具體實例，它實體擁有特定的數據，且能對其進行特定的操作。從代碼上看，類是對象的模板，通過這個模板可以創建許多相似的對象。

在上節的示例中，我們只創建了一個對象，名爲 object。可以調用其 Hello() 方法，效果是根據其名字打印一條歡迎致辭。如果要表達另一個對象，一個笨辦法是修改其屬性的值。例如:

```
: call object.Hello()    |" --> Hello bob!!!
: let object.name = 'ann'
: call object.Hello()    |" --> Hello ann!!!
```

但是，這仍然只實際存在了一個對象。如果在程序中要求同時存在兩個相似的對象，那該如何? 也很容易想到，只要克隆一個對象，再修改有差異的數據即可。當然了，你不用在源代碼上覆制粘貼一遍對 object 的定義，只要調用內置函數 copy()。因爲有關該對象的所有東西都已經在 object 中了，在 VimL 看來它就是一個字典變量而已。如:

```
: let another_object = copy(object)    |" 或 deepcopy(object)
: let another_object.name = 'ann'
: call another_object.Hello()    |" --> Hello ann!!!
```

不過要注意的是，由於當初在定義 object 時，預設了一個字典成員屬性 data。如果用 copy(object) 淺拷貝，則新對象 another_object 與原對象 object 將共享一份 data 數據，即如果改變了一個對象的 data 屬性，另一個對象的 data 屬性也將改變。如果設計需求要求它們相互獨立，則應該用 deepcopy(object) 深拷貝方法。

以上用法可行，但不盡合理。因爲在實際程序運行中，object 的狀態經常變化，在一個時刻由 object 複製出來的對象與另一個時刻複製的結果不盡相同，且不可預期。那麼就換一種思路。可以先定義一個特殊的字典對象，其主要作用只是用來 “生孩子”，克隆出其他對象。即它只預定義（設計）必要的屬性名稱，及提供通用的初值。當在程序中實際有使用對象的需求時，再複製它創建一個新對象。於是，這個特殊的字典，就可充當類的作用。類也是一個對象，不妨稱之爲類對象。

於是，可修改上節的代碼如下:

```
let class = {}
let class.name = ''
let class.desc = 'a sample class'
let class.value = 0
let class.data = {}

function! class.EndHi() abort dict
    return '!!!'
endfunction

function! class.Hello() abort dict
    echo 'Hello ' . self.name . self.EndHi()
endfunction

: let obj1 = deepcopy(class)
: let obj1.name = 'ann'
: call obj1.Hello()    |" --> Hello ann!!!
: let obj2 = deepcopy(class)
: let obj2.name = 'bob'
: call obj2.Hello()    |" --> Hello bob!!!
```

這裏，先定義了一個類對象，取名爲 class。其數據屬性與方法函數定義與上節的 object 幾乎一樣，不過是換了字典變量名而已。另外，既然 class 字典是被設計爲當作類的，不是實際的對象實例，那它的 name 屬性最好留空。當然了，取名爲 noname 之類的特殊字符串也許也可以，不過用空字符串當作初值足夠了，且節省空間。然後，使用這個類的代碼就簡單了，由 deepcopy() 創建新對象，然後通過對象訪問屬性，調用方法。

這就是 VimL 所能提供的面對象支持，用很簡單的模型也幾乎能模擬類與對象的行爲。當然了，它並不十分完美，畢竟 VimL 設計之初就沒打算（似乎也沒必要）設計爲面向對象的語言。如果在命令輸入以下命令查看這幾個字典對象的內部結構：

```
: echo class
: echo obj1
: echo obj2
```

可以發現，對象 obj1，obj2 與類 class 具有完全一樣的鍵數量（當然了，因爲是用 copy 複製的呀）。VimL 本身就沒有在對象 obj1 與類 class 之間建立任何聯繫，是我們（程序員）自己用全複製的方式使每個對象獲得類中的屬性與方法。每個對象應該有自己獨立的數據，這很容易理解與接受。但是，每個對象都仍然保存着應該是通用的方法，這似乎就有些浪費了。幸好，這只是保存函數引用，不管方法函數定義得多複雜，每個函數引用都

固定在佔用很少的空間。不過，蚊子再小也是肉，如果一個類中定義了很多方法，然後要創建（複製）很多對象，那這些函數引用的冗余也是很可觀的浪費了。

另外，直接裸用 copy() 或 deepcopy() 創建新對象，似乎還是太粗糙了。如果數據屬性較多，還得逐一賦上自己的值，這寫起來就比較麻煩。因此，可以再提煉一下，封裝一個創建新對象的方法，如：

```
function! class.new(name) abort dict
    let object = deepcopy(self)
    let object.name = a:name
    return object
endfunction

: let obj3 = class.new('Ann')
: call obj3.Hello()    |" --> Hello Ann!!!
: let obj4 = class.new('Bob')
: call obj4.Hello()    |" --> Hello Bob!!!
```

不過，仍然有上面提及的小問題，class 中的 new() 方法也會被複制到每個新建的對象如 obj3 與 obj4 中。若說在類中提供一個 new() 方法很有意義，那在對象實例中也混入 new() 方法就頗有點奇葩了，只能選擇性忽略，不用它就假裝它不存在。當然了，如果只是想封裝“創建對象”這個功能，也可用其他方式迴避，這在後文再敘。

這裏要再提醒一點是，由於 new() 方法是後來添加進去 class 類對象中的。在這之前創建的 obj1，obj2 對象是基於原來的類定義複製的，所以它並不會有 new() 方法，在這之後創建的 obj3，obj4 纔有該方法。如果在之後給類對象添加新的屬性或（實用）方法，也將呈現這種行爲，原來的舊對象實例並不能自動獲得新屬性或方法。有時固然可以有意利用這種動態修改類定義的靈活特性，但更多的時候應該是注意避免這種無意的陷阱。這也再次說明了，VimL 語法本身並不能保證對象與類之間的任何聯繫，其間的聯繫都是“人爲的假想”，或者說是程序員的設計。

複製字典也是繼承

上文已經講了，通過在字典中同時保存數據與函數（引用），就基本能實現（模擬）面向對象的封裝特徵。然後，面向對象的另一個重要特徵，繼承，該如何實現呢？其實一句話點破也很簡單，也就是用 copy() 複製，複製，再複製。

接上節的例子，我們打算從 class 類中繼承兩個子類 CSubA 與 CSubB，示例代碼如下：

```
let CSubA = deepcopy(class)
function! CSubA.EndHi() abort dict
    return '$$$'
```

```
endfunction

let CSubB = deepcopy(class)
function! CSubB.EndHi() abort dict
    return '###'
endfunction

: let obj5 = CSubA.new('Ann')
: call obj5.Hello()    |" --> Hello Ann$$$
: let obj6 = CSubB.new('Bob')
: call obj6.Hello()    |" --> Hello Bob###
```

在這兩個子類中，我們只重寫覆蓋了 EndHi() 方法，讓每個子類使用不同的語氣符號後綴。而基類 class 中的 new() 方法與 Hello() 方法，自動被繼承（就是複製啦）。其實 EndHi() 方法也是先複製繼承了，只是立刻被覆蓋了（所以必須用 ':function!' 命令，加歎號）。在使用時，也就可以直接用子類調用 new() 方法創建子類對象，再子類對象調用 Hello() 方法。

至於面向對象的多態特徵，對於弱類型腳本語言而言，只要實現了繼承的差異化與特例化，是天然支持多態的。例如上面的 obj5 與 obj6 雖屬於不同類型，但可直接放在一個集合（如列表）中，然後調用統一的方法：

```
: let lsObject = [obj5, obj6]
: for obj in lsObject
:     call obj.Hello()
: endfor
```

但是對於其他強類型語言（如 C++），卻不能直接將不同類型的 obj5 與 obj6 放在同一個數組中，才需要其他語法細節來支持多態的用法。

小結

VimL 提供的字典數據結構，允許程序員寫出面向對象風格的程序代碼。在字典內，可同時保存數據與函數（引用），並且在這種函數中可以用關鍵字 self 訪問字典本身，因而可以將字典視爲一個對象。類、繼承子類與實例化對象，都可以簡單通過複製字典來達成。只是這種全複製的方式，效率未必高，因爲會將類中的方法即函數引用都複製到子類與實例中，即使函數引用所佔空間很小，也會造成一定的浪費。然而，從另一方面想，光腳的不怕穿鞋的，VimL 本來就不講究效率，這也不必太糾結。幾乎所有的語言，面向對象設計在給程序員帶來友好的同時，都會有一定的實現效率的代價交換。

7.3 自定義類的組織管理

在上一節已經講敘瞭如何利用 VimL 語法實現面向對象編程的基本原理，本節進一步討論在實踐中如何更好地使用 VimL 面向對象編程。關鍵是如何定義類與使用類，如何管理與組織類代碼使之更整潔。因爲從某種意義講，面向對象並不是什麼新的編程技術，而是抽象思考問題的哲學，以及代碼管理的方法論。

筆者在 github 託管了一個有關 VimL 面向對象編程的項目 vimloo，可作爲一個實現範例。本節就介紹這個 vimloo 項目的基本思路，不過該項目代碼有可能繼續更新維護與優化，故本節教程所採用的示例代碼爲求簡單，不盡與實際項目相同。

每個類獨立於一個自動加載文件

在上一節的示例代碼中，我們定義了一個名爲 class 的類。因爲彼時只關注實現原理，並未指定相關代碼應保存何處。你可以放在任一個腳本中，甚至也可以粘貼入命令行，也能起到演示之用。

如果你想用 VimL 實現一個規劃不太大的（插件）功能，又想用到字典的對象特徵，想在單文件中實現全部（或大部分）功能，那麼也着實可以就像是上節的示例那樣，在單文件中定義類然後使用類。但是，既然想到要用面向對象的設計，那麼一般地每個類都應該是相對獨立完整的功能單元。這時，將類的定義代碼提取出來放在獨立的文件中就更合適了，這也可以達到隱藏類實現細節的目的，在其他需要使用對象的地方，只需創建相應類的對象，調用該類對象所支持的方法即可。

簡言之，要區分類的實現者與使用者（儘管很多時候這是同一個程序員的工作）。在 VimL 中，如果要將類的定義代碼單獨存於一個文件中，最適合的地方應該就是 autoload/ 子目錄下的自動加載文件了。因爲它可以讓用戶從任意地方調用，並且只在真正需要用到時才加載類定義代碼。

於是，將上節的 class 類定義稍作修改，保存於某個 &rtp （如 ~/.vim）的 autoload/class.vim 文件中：

```
1  " File: ~/.vim/autoload/class.vim
2
3  let s:class = {}
4  let s:class.name = 'class'
5  let s:class.version = 1
6
7  function! s:class.string() abort dict
8      return self.name
9  endfunction
10
11 function! s:class.number() abort dict
12     return self.version
```

```
13 endfunction
14
15 function! s:class.disp() abort dict
16     echo self.string() . ':' . self.number()
17 endfunction
```

主要是將定義的類（字典）名字改爲 s:class，使之成爲局部於腳本的變量。這樣在不同文件中定義的不同類也都能用相同的字面名字 s:class 而互不衝突。該變量名的選用是任意的，在不同類文件中選用不同變量名也可以，只要在隨後定義類的屬性與方法也都用相應的字典變量名即可。但這裏的建議是，爲求風格統一，每個類文件定義的類字典變量都取名爲 s:class。

在這個 class.vim 中定義的類沒打算做什麼實際工作，因此只（貌似隨意地）定義了兩個屬性與幾個方法。當然，你也可以將 string() 與 number() 方法想象爲類型轉換方法，用於在必要時如何將一個對象轉爲字符串或數字的表示法。

使用自動加載函數處理類方法

現在 class.vim 文件中定義的 s:class 類只能在該文件中訪問，這顯然是不夠的。爲了達到分離類定義與類使用的設計原意，我們還得在 class.vim 提供一些公有接口讓外界使用類。自動加載函數就是一個很好的選擇，因爲它既是全局函數，又通過 # 前綴限定了“僞作用域”。例如，添加以下函數：

```
1  " File: ~/.vim/autoload/class.vim
2
3  function! class#class() abort
4      return s:class
5  endfunction
6
7  function! class#new() abort
8     let l:obj = copy(s:class)
9     return l:obj
10 endfunction
11
12 function! class#isobject(that) abort
13     return a:that.name ==# s:class.name
14 endfunction
```

先看 class#class() 這個略有奇怪的函數命名。class# 前綴部分是對應 class.vim 文件名路徑的，class() 可認爲是該函數的基礎名字。它的作用很簡單（也很關鍵），就是返回當前文件定義的類 s:class，使外界有個途徑能使用這個類。這就是個取值函數，也可命名爲

getclass() 或許可更易理解。

class#new() 函數就是用於創建一個新對象。我們使用一個類時，第一步往往就是新建對象，這就只要調用 class#new() 就可以了。如果之前尚未加載類 class 的定義，就會按自動加載機制加載 class.vim，也就完成其內 s:class 的定義。普通用戶一般情況下根本用不到 class#class() 獲取 s:class 的定義，除非想動態修改類定義（慎重）。如果真的想向用戶完全隱藏類定義，不提供 class#class() 函數即可，只提供 class#new() 讓用戶能創建對象好了。

所以纔將創建對象的函數定義爲 class#new() 而非像上節那樣的方法 s:class.new()，讓用戶直接上手創建對象，而不必關心類定義是否已加載。其次也是由於 VimL 只能按複製式創建對象，如果把 s:class.new() 方法也複製到對象中，是很沒必要的，甚至還可能被誤用。

至於 class#isobject() ，用於判斷一個對象是否屬於本文件所定義的類。在某些應用中，先作類型判斷是有意義的甚至是必要的。這裏暫且先用類的 name 屬性來標記一個類，因此爲了保證類名的唯一性，name 屬性的取值也按自動加載函數的規則取文件名路徑（即如 class#class() 函數的前綴部分）。如果在某個深層子目錄中定義的類，如 autoload/topic/-subject.vim 文件內定義的 s:class 類名屬性就應該是 topic#subject。當然了，另有一個建議，由於 VimL 的大多數腳本都未必是類定義文件，爲了更明確表示它是個類文件，可將更多實用的類都統一放在 class/ 子目錄下，如 autoload/class/topic/subject.vim，如果其類名就是 class#topic#subject。嚴格地講，class#isobject() 要穩健地執行，還應判斷所傳入的參數 a:that 是否字典類型，以及是否有 name 這個屬性。

然後，可以根據需要設計更多的函數。這有兩種選擇，如果是操作對象的方法，應存入 s:class 字典，如 s:class.method()。如果它不適合用作對象的方法屬性，而着重與類型有關，可定義爲自動加載函數，如 class#func()。

區分類屬性與對象屬性

從前面的章節討論中，我們意識到類屬性與對象屬性可以是兩個不同的概念，這是值得優化的一個方向。尤其是 VimL 中若用簡單粗暴的全複製方式創建對象，把那些通用的屬性複製到每個對象中，顯然是個浪費。例如上一小節的類名屬性 name，尤其是深層目錄的類文件，像 class#topic#subject 這樣的字符串已經不短了，每創建一個新對象都保存這樣一個屬性值，似乎很不值了。

但另一方面，在類定義字典中保存類名屬性也是有意義的，因其關聯了文件路徑，也可據此間接調用方式文件內的自動加載函數。所以，最好是能限定類名屬性不被複制到新建對象中。因此爲了區分，約定將類屬性的命名加兩個下劃線，如 __name__。這樣，某些具體的對象也可能需要自己的 name 屬性，也不致鍵名衝突。

按這種思路，我們再試寫另一個類文件：

```
1 " File: ~/.vim/autoload/class/subclass.vim
2
```

```
3  let s:class = {}
4  let s:class._name_ = 'class#subclass'
5  let s:class._version_ = 1
6
7  " Todo: 其他對象屬性預設
8  let s:class.value = 0
9
10 function! class#subclass#class() abort
11     return s:class
12 endfunction
13
14 function! class#subclass#new() abort
15    let l:obj = Copy(s:class) " Todo: 另外定義的特殊拷貝函數
16    return l:obj
17 endfunction
```

之前定義在 autoload/class.vim 文件中名爲 class 的類，不妨當作整個自定義 VimL 類系統的通用基類。在實際工作中一般不會直接用到 class 類及其實例對象。所以我們開始設計實際可用的子類，建議將所有實用類歸於 class/ 子目錄下。以上也僅是個說明示例，故類名簡單取爲 subclass，按自動加載機制，其全名則是 class#subclass 。

這個類文件的基本框架與之前類似，只不過將原來的類屬性改名爲 _name_ 與 _version_ 。屬於該類的對象的屬性名，不加下劃線，比如 value。然後創建對象的 #new() 函數，顯然不能直接用 copy() 或 deepcopy() 內置函數了。這個輔助的特殊複製函數需要另外實現，不過將其命名爲 Copy() 或 SpecialCopy() 就顯得有點蠢了。聯想到之前的 class#new() 函數，既然一般沒必要創建 class 頂層基類的實例對象，不妨將 class.vim 內定義的函數改爲公共基礎設施函數。於是修改如下：

```
1  " File: ~/.vim/autoload/class/subclass.vim
2
3  function! class#subclass#new(...) abort
4     let l:obj = class#new(s:class, a:000)
5     return l:obj
6  endfunction
```

這裏，只是將當前文件定義的類 s:class 與任意參數 a:000 傳給 class#new() 基礎設施函數，然後也是返回所創建的對象。至於 class#new() 的具體實現，略複雜，請參考 vimloo 項目的 autoload/class.vim。這裏只說明它主要做的幾件事：

一是分析 s:class 的鍵，過濾掉帶下劃線前後綴的屬性名，只把普通屬性複製到對象實例中。如上例的 class#subclass 類，由 #new() 創建出的對象只有 value 屬性。

二是給每個新建對象添加唯一一個特殊屬性，名爲 _class_ ，就是對 s:class 的引用。這樣每個對象都能知道自己所屬的類了，在有必要時可訪問這個類字典獲得其他信息。而且保存類字典的引用，比保存類名字符串在安全性與效率性上都好得多。然後，判斷一個對象是否屬於本類的函數也能利用該屬性，可大約修改如下：

```
" File: ~/.vim/autoload/class/subclass.vim

function! class#subclass#isobject(that) abort
    " is 是操作符，相當於 == 用於比較相同的引用
    return type(a:that) = type({})
        / && get(a:that, '_class_', {}) is s:class
endfunction
```

其實還有第三個隱藏事件，這隻在每個類創建第一個對象時發生。爲了避免每次創建對象都要作第一步的分析過濾 s:class 的鍵名，class#new() 會在第一次記憶這個結果，保存在一個特殊鍵 s:class._object_ 中。這是向用戶隱藏的第一個實例，用戶新建使用到的實例是直接從這個實例深拷貝的（deepcopy()）。我們可以將其視爲這個類的“長子”，是其他實際干活的小弟們的楷模。

控制繼承與多層繼承

然後討論 vimloo 項目對繼承的實現。首先不要驚訝於命名學上的選用。因爲前文已經說明，繼承與實例化一樣底層都是通過複製實現的。既然創建新對象是用 #new() 函數，那麼創建新子類就用個相對的單詞 #old() 。

假設要從 subclass 繼承一個類 subsubclass，類文件保存於 class/subsubclass.vim。當然你也可保存於 class/subclasss/subsubclass.vim 文件中，只是名字略長。這裏要指出的是，文件系統的目錄層次，未必要強求與類的繼承鏈一一對應，那也會有其他麻煩，僅從文件管理角度看，將相關主題的類文件放在一個目錄中就能接受了。

要實現這個繼承關係，有兩點需要改動。一是在 subsubclass.vim 中創建 s:class 時不再初始化爲空字典，而是調用 subclass#old() 返回的字典；二就是要在 subclass.vim 中實現 subclass#old() 函數，描述如何將自己這個類繼承（複製）給子類。代碼框架如下：

```
" File: ~/.vim/autoload/class/subsubclass.vim

let s:class = subclass#old()
let s:class._name_ = 'class#subsubclass'
let s:class._version_ = 1

" Todo: 其他類屬性與方法
function! class#subsubclass#new(...) abort
```

```
    let l:obj = class#new(s:class, a:000)
    return l:obj
endfunction
```

```
" File: ~/.vim/autoload/class/subclass.vim

" 其他沿用，添加 #old() 方法
function! class#subclass#old(...) abort
    let l:class = class#old(s:class)
    return l:class
endfunction
```

可見 subsubclass.vim 的類定義框架與之前的 subclass.vim 很是類似，只有第一行初始化 s:class 的不同。甚至創建對象的 #new() 方法的寫法也完全一樣，因爲把複製的細節都提煉到 class#new() 這個通用設施上了。用戶可直接上手調用 class#subsubclass#new() 方法創建對象，按 VimL 自動加載機制，subsubclass.vim、subclass.vim 與 class.vim 這三個腳本文件都會觸發加載。

至於繼承函數 class#subclass#old() 與實例化函數 class#subclass#old() 也類似，將複製的細節委託通用的 class#old() 函數處理。它也是分析過濾 s:class 的鍵，將必要的鍵複製給子類，並在子類字典中添加一個特殊鍵 _mother_ 引用自身類字典。（具體實現代碼就不帖了，看 vimloo 項目源碼）

如果要讓 subclass 繼承自 class，也可修改 subclass.vim 中對 s:class 的創建語句 let s:class = class#old()。因爲 class#new() 與 class#old() 函數接收可變參數，一般將其第一個參數視爲類定義字典，即其他類文件中的 s:class，當然也可以是類名字符串，根據類名可獲取其 s:class 字典；如果沒有參數時，就用 class.vim 文件本身的 s:class 類字典。不過，由於在 class.vim 的 s:class 在實踐中實在乏善可陳，在第二版（_version_ = 2）時，無參調用 class#old() 會快速返回空字典 。自定義的頂層類沒有母類（基類），或 _mother_ 屬性爲空。

因此，vimloo 實現的類體系，可類比“母系社會”來理解。從一個母類中有兩種繁衍，“女兒”是子類，主要用途就是繼續繁衍；“兒子”是實例化對象，就是用來實際工作干活的。子類中通過 _mother_ 屬性記錄母類的聯繫，實例中是 _class_ 屬性。由於實際工作中可能需要許多同質的實例對象，故而還設置了一個隱藏的 _object_ 長子監管。這套機制用於描繪單繼承應該足夠清晰易懂。

能用單繼承解決的問題，儘量避免多重繼承。不過 vimloo 也實現了多重繼承的支持。每個類的 _mother_ 屬性雖然只記錄了唯一的母類，但也允許有其他基類，有兩種“其他基類”。一種叫 _master_（意爲“師父”），只繼承其方法，不繼承其數據；另一種叫 _father_（意爲“父親”），只繼承其數據，不繼承其方法。每個類的 _master_ 與 _father_ 屬性（如果有），都是數組，即可以是多個來自其他類文件定義的 s:class。只不過這些“其他基類”的

屬性，都不會直接導入當前文件的 s:class 中，只有當創建對象實例時（如 s:class.__object__ 長子），纔會分析這些類的鍵名，將必要的鍵複製下來。

也可以通過形象的比喻來理解這個模型。如一位母親撫育孩子，額外聘請多位老師教孩子其他技藝，這是可理解的（相當於某些語言的接口方法），不過母親本身未必要掌握這些技能，她的目的是孩子們能學會就可以了。當然了，另一方面，也允許多個“父親”，這思想有點危險啊，最好避而不用吧。

構造函數與析構函數

重新審視一下創建對象的 #new() 方法，其流程應該要包含以下三步工作：

1. 複製類字典
2. 初始化對象屬性
3. 返回對象

其中，第一步與第三步的工作，對於每個類而言，都幾乎是一樣的，所以在 vimloo 中將其提煉爲 class#new() 函數，可爲每個自定義類處理通用事務。但是第二步的初始化，顯然是每個類有獨立需求的。因此，建議每個類文件再寫個 #ctor() 函數專司初始化，這就叫做構造函數。

仍以上文的 subclass 爲例，將其創建函數與構造函數並列展示如下：

```
" File: ~/.vim/autoload/class/subclass.vim

function! class#subclass#new(...) abort
   let l:obj = class#new(s:class, a:000)
   return l:obj
endfunction

function! class#subclass#ctor(this, ...) abort
    if a:0 > 0
        let a:this.value = a:1
    endif
endfunction
```

理論上，#ctor() 函數內的初始化代碼插入到 #new() 函數中也是可以的。不過爲了保持 #new() 函數的簡單統一，同時爲了支持其他間接創建對象的需要，故將構造函數 #ctor() 獨立出來。需要注意的是，#ctor() 函數不是由當前類文件的 #new() 函數直接調用的，而是間接由通用的 class#new() 函數調用。不過可變參數... 的意義在這兩個函數之間保持一致，即 #ctor() 內的 a:1 與 #new() 內的 a:1 是相同意義的參數。在構造函數 #ctor() 中，對象已經被創建出來，第一個參數 a:this 就代表這個剛創建的對象。構造函數一般不由用戶直接調用，也不必返回值，只要在創建函數 #new() 中返回對象即可。

一般情況下，在自定義類文件中，建議同時提供創建函數與構造函數，各司其職。但是構造函數不是必須的，尤其是對象屬性很少，或能接受每個對象都採用相同的初始值。甚至創建函數也不是必須的，因爲也能從通用的 class#new() 函數中創建指定類的對象。例如，以下兩個語句是等效的：

```
: let obj = class#subclass#new(100)
: let obj = class#new('class#subclass', [100])
```

顯然，使用類文件自己特定版本的 #new() 函數創建對象更簡潔，意義更明確。不過通用的 class#new() 函數也適用於在程序運行需要動態創建不同類別的對象的情況。如果傳入的第一個參數是類名字符串，則相應的類文件中必須定義 #class() 函數（上例就是 class#subclass#class()）才能獲取其類定義 s:class。此外，要讓 class#new() 能正確調用構造函數，也依賴於類字典 s:class 保存了類名屬性 __name__。

對於子類的構造函數，寫起來略爲複雜些。因爲你肯定期望能複用基類（母類）的構造函數初始化繼承自母類的那部分數據屬性。class.vim 提供了一個 class#Suctor() 函數用於獲取一個類的母類的構造函數（引用）。於是 subsubclass 的構造函數可寫成如下形式：

```
1 " File: ~/.vim/autoload/class/subsubclass.vim
2 function! class#subsubclass#ctor(this, ...) abort
3     let l:Suctor = class#Suctor(s:class)
4     call call(l:Suctor, extend([a:this], a:000))
5     " Todo: 子類的其他對象屬性初始化
6 endfunction
```

其中，call() 內置函數的用法不算簡單，請參考文檔 ':h call()'。如果你確知母類的構造函數沒有做什麼實質性的初始化工作（甚至未提供構造函數），也可以省去調用母類構造函數的步驟。如果硬編碼調用母類的構造函數，如 class#subclass#ctor() ，也不是不可以，但顯然太過僵硬了，且寫法上也未必比利用 class#Suctor() 省多少。在上例中，直接將所有的參數 a:000 傳給母類的構造函數處理。在實踐中，可能只需要部分參數傳給母類，如果這部分參數正好是可變參數的前面幾個，那麼直接傳 a:000 也可能是正常的。在其他其他情況下，可能要對參數作某些預處理再傳給母類的構造函數。

在那些沒有自動回收垃圾機制的面向對象語言（如 C++）中，與構造函數相應地，還有析構函數。VimL 腳本語言顯然是能自動回收垃圾的，不須由程序員作此負擔。不過 VimL 在處理有環引用（如雙向鏈表、樹、圖等複雜結構）中，垃圾回收會有滯後。爲此，也可以在自定義類文件中寫個 "析構函數"，命名爲 #dector()，用於打斷對象內部的環引用。當確實用不到一個對象時（往往是在函數末尾），調用 class#delete(object)，它會自動調用相應類文件的 #dector() 方法，然後當這個對象離開作用域時，就能立即被回收了。vim 也有個內置的函數 garbagecollect() 可觸發立即回收垃圾，但它可能要用到搜索判斷環引用的複雜算法。如程序員能幫它的回收機制打斷環引用，也應是善事，儘管這是可選的，不是必須的。

類的外包與簡化使用

有了以上討論的 vimloo 提供的面向對象功能，我們就能根據具體的功能需要，設計自定義的類（體系）了，然後創建對象完成實際的工作。

不過還有個小問題，就是類名可能太長，書寫不便。假如有這麼個類，全名是 class#long#path#topic#subject。用戶在使用這個類時，每次創建對象都得調用 class#long#path#topic#subject#new() 函數。這已經算麻煩的了，如果以後想重構，想對類重命名或移動存放目錄路徑，那每個創建對象的地方都還得作相應修改，那就不僅麻煩，也更易遺漏出錯了。

爲此，vimloo 再提供一個 class#use() 函數。先直接看用法示例：

```
" File: ~/.vim/autoload/class/long/path/topic/subject.vim

" 正常類定義，略

function! class#long#path#topic#subject#use(...) abort
    return class#use(s:class, a:000)
endfunction
```

```
" File: ~/.vim/vimllearn/useclass.vim

let s:CPack = class#long#path#topic#subject#use()
" 或
" let s:CPack = class#use('class#long#path#topic#subject')

function! s:foo() abort
    let l:obj = s:CPack.new()
    " Todo:
endfunction

function! s:bar() abort
    let l:obj = s:CPack.new()
    " Todo:
endfunction
```

簡言之，class#use() 創建會創建一個字典，默認情況下有以下幾個鍵：

- class：就是引用在類文件中定義的類字典 s:class
- new ： 函數引用，相關類文件的創建函數 #new()
- isobject ： 函數引用，相關類文件的創建函數 #isobject()

就是將某個類定義及兩個最重要的自動加載函數（的引用）打包在另一個字典中，可以提供額外參數（函數名列表，不含 # 路徑前綴）指定打包其他的自動加載函數，但 class

是不需要指定，必然被打包在其內的。由於這僅是作了一層簡單的包裝，提供給外部使用，故簡稱爲“外包”機制。注意類的方法（如 s:class.method()）是不需要外包的，因爲那是通過之後創建的實例對象訪問的。

通過這種外包，用戶代碼就可大爲簡化了。例如可以在腳本開始將要用到的類的外包保存在一個腳本局部變量，如 s:CPack，然後在該腳本內就可以用 s:CPack.new() 創建該類的對象了。這是自動加載函數的引用，同樣可以觸發相關類文件的自動加載。如果此後類名發生了修改，或者就是想試用另一個類，也只要修改開始的一處代碼而已。甚至在創建子類時，也可以利用外包書寫，如：

```
let s:CPack = class#long#path#topic#subject#use()
let s:class = class#old(s:CPack.class)
" 等效於
" let s:class = class#long#path#topic#subject#old()
```

另外要提示的是，class#use() 函數會記錄已經被外包使用的類。所以在正常運行時，每個類只會創建一個外包，在多個腳本中使用同一個類的外包時，並不會增加額外的開銷。

類文件框架自動生成

從以上內容可感知，創建一個自定義類文件，有着大致相似的框架，主要包含以下幾部分內容：

- 創建 s:class 字典，可以是簡單的空字典或繼承其他類；
- 爲 s:class 增加數據屬性鍵，可用初始值約定數據類型；
- 爲 s:class 創建字典函數，用作類的方法；
- 提供一些必要的自動加載函數。

爲了節省鍵盤錄入字符的工作，vimloo 也提供了一些命令，用於根據模板文件生成類定義文件的基本框架。這可節省 VimL 類開發者的大量工作，通過命令生成基礎代碼（甚至可以再自定義映射，一鍵生成）後，只要再填充必要的類定義實現即可。

- :ClassNew {name} 當前目錄在某個 autoload/ 或其子目錄時可用，提供一個文件名參數，將新建一個 '.vim' 文件，並根據該文件名創建一個類。
- :ClassAdd 當正在編輯 autoload/ 或其子目錄下的某個 '.vim' 文件時，用該命令向當前文件添加一個類定義。
- :ClassPart {option} 與:ClassAdd 類似，但只根據選項生成部分代碼，而非全部代碼，用於補遺。

類定義的框架模板文件位於 vimloo 項目的 autoload/tempclass.vim，這也是一個符合 VimL 語法的腳本，同時也是個五髒俱全的類定義文件。該文件的每個段落開始有行註釋，註釋行末尾是類似 -x 的選項字符串，其中若小寫字母表示默認生成這段代碼，大寫字母表示不生成這段代碼。但以上命令可附加額外選項覆蓋默認行爲，多個選項字母拼在一起當作一個參數傳入。

若使用時還遇到疑問，請參考 vimloo 項目的說明文檔或幫助文檔。

第八章　VimL 異步編程特性

8.1　异步工作簡介

異步機制是 vim8 版本引入的新機制，准確地說，是從 7.4 某個補丁開始引入，不過在 vim8 完善並正式發佈。這一全新特性使得 vim 直接跳升一大版本號，可見意義非凡。

同步工作可能的問題

要理解異步的特性，不妨先回顧下在此之前只能同步工作的情況，會遭遇哪些不便。

比如要從一個目錄下的文本文件中查找某個字符串，我們知道（在 unix 系統中）直接有個 grep 工具可用。而在運行着的 vim 中，也可以通過‘:!grep ...’命令調用系統的 grep 工具。但是用‘:!’執行外部命令的話，會臨時切回啓動當前 vim 的終端，外部命令的輸出在該終端上；當外部命令經過或長或短的時間完成後，還需要等用戶按回車確認纔回到 vim 正常的用戶界面。如果是 windows 系統的 gVim，‘:!’執行外部命令則會彈出 cmd 黑框，展示外部命令的輸出，也需要由用戶確認關閉該黑窗才能回到 gVim 編輯窗口。

顯然按種方式，在運行外部命令的同時，在回到 vim 界面之前，vim 對用戶而言是停止工作的，比如用戶暫時無法操縱 vim 進行編輯工作。vim 也有個類似的內部命令‘:vimgrep’用於在多文件中搜索字符串，並將結果輸出在 quickfix 窗口。運行該命令不會切回 shell 終端，與‘:!grep’很有些不同。但是，如果待搜索的文件很多，尤其是類似 **/* 的遞歸所有子目錄的文件搜索時，‘:vimgrep’命令完成搜索也可能很慢，需要等待一段時間才能完成搜索。在等待的這段時間內，雖然仍然停留在 vim 界面，但 vim 也好像停止了與用戶的交互工作，譬如按 j k 不見得會移動光標。事實上 vim 還是監測到你按了 j k 鍵，只不過要等‘:vimgrep’這個慢命令完成後纔會響應後續按鍵。簡單地說，就可能造成明顯卡頓。

這就是舊版本 vim 按同步工作方式可能出現的問題。你可以將 vim 編輯器想象爲一個單線程的無限循環程序，等待着用戶的按鍵，並立即根據按鍵命令處理工作。正常情況下 vim 響應用戶按鍵命令是極快的，所以用戶感覺很流暢。因爲正常人類的擊鍵速度在計算機程序看來都太慢了，vim 在大部分時間裏都是在等待用戶擊鍵的。但是當用戶試圖讓 vim 執行某些“能感覺出來慢”的命令時，問題就浮現了，影響用戶體驗。

如果上面的‘:vimgrep’命令沒讓你感覺到慢，可以用 VimL 定義如下的慢函數：

```
1 function! SlowWork()
2     sleep 5
```

```
3       echo "done!!"
4   endfunction
```

然後在命令行輸入‘:call SlowWork()’並回車，你應該就能感覺到 vim 明顯卡頓了。在此期間若按幾次 j，也要等該函數返回才能發現光標移動。此外，你也可以試試用 while 1 | endwhile 定義一個無限循環函數，調用時會令 vim 完全停止響應，如此請用 Ctrl-C 強制結束當前命令，回到 vim 的正常工作狀態來。

異步工作想解決的問題

顯然，vim8 引入的異步機制，就是試圖解決（或部分解決、緩和）上述同步模型中出現的“慢命令卡頓”問題。當然它也不是直接重定義優化原來命令的工作方式，因爲兼容舊習慣也是 vim 的傳統。所以，在 vim8 中，類似‘:!grep’或‘:vimgrep’命令，該怎麼慢還怎麼慢，它真正想優化的類似 system(‘grep’) 函數的工作方式。

system(‘grep’) 與‘:!grep’的相同之處在於都是調用外部命令（系統可執行程序），只不過調用 system() 函數不會切到 shell 終端，仍停留在 vim 界面。所調用的外部命令的輸出會被 system() 函數所捕獲，可以保存在 VimL 變量中，供腳本後續使用。如果該外部命令執行時間較長，vim 用戶仍會感到停止響應或卡頓。

然後在 vim8 中，就提供了另一套不叫 system() 名字的函數，用於執行外部命令。vim 不再等待外部命令結束，而是立即返回給用戶，可以立即接着響應用戶按鍵。等外部命令終於結束了，vim 再調用一個回調函數處理結果。

開啓異步工作的具體函數與用法，留待下一節詳細介紹。不過你應該能感覺與估計到，這個異步編程模型比本書之前介紹的同步編程模型要複雜些。並且在監測外部命令結束時准備回調也必然有其他開銷，所以異步也不宜濫用，只適合在（可預期）比較耗時的外部命令上。如果只是簡單的可以快速完成的外部命令，仍用原來的 system() 函數完成工作即可。

另外要提及的是，目前 vim8 版本的異步機制，也只能將外部命令以異步的方式開啓，並不能用異步的方式執行內部命令。也就是說，不論是 vim 內置的命令（及常規函數），還是用 VimL 寫的自定義命令（函數），都仍只能按原來的同步方式執行，暫無異步用法。

異步機制帶來的 vim 新特性簡介

vim8 提供異步機制後，可以據此實現很多新特性。比如內置終端（從 vim8.1 版本開始支持）。在命令行執行‘:terminal’就能打開一個新窗口，體驗一下內置終端。在這個特殊的 vim 新窗口中，就相當於運行着一個 shell ，可以像系統 shell 一樣執行任何命令，甚至也可以在此又運行一個 vim （不過一般情況下不建議這麼玩）。用窗口切換快捷鍵 Ctrl-w 可以回到之前的普通 vim 窗口，正常操作 vim 進行編輯工作。

也就是說，內嵌終端正是異步運行的，並不中斷 vim 本來的編輯工作。相比在這個功能出現之前，用‘:shell’命令打開的子終端，就會切出 vim 界面，只能在那個子終端中工作，必須在那執行 $ exit 退出子終端，才能回到 vim 。

關於內置終端的詳細用法請參考 ':help terminal', 在那文檔中還介紹了在 vim 中 "嵌入" gdb 調試 vim 本身的示例。表明內置終端功能其實不止能執行一個 shell ，還適於執行其他任何交互程序，例如 python 解釋器，mysql 客戶端，gdb 調試器等。

不過本章不是介紹 vim 的這類新特性，而是側重介紹 VimL 腳本編程中如何使用這個異步機制，據此可以完成之前的腳本無法完成的工作，或優化某些插件功能。

異步編程的簡單運用：定時器

讓我們先看一個簡單的例子來體驗下異步編程的風格，定時器（請確認 vim 編譯包含 +timers 特性）。將上文按傳統同步風格定義的 SlowWork() 函數重新改寫如下：

```
function! SlowWork()
    call timer_start(5*1000, 'DoneWork')
endfunction

function! DoneWork(timer)
    echo "done!!"
endfunction

call SlowWork()
```

現在再調用 SlowWork() 函數時就不會 "暫停" 5 秒了，該調用立即返回，用戶可如常操作 vim 。大約過了 5 秒後，函數 DoneWork() 被調用，顯示 "done!!"。

這裏的關鍵是在 SlowWork() 中用 timer_start() 啓用了一個定時器。參數一是時間，單位毫秒；參數二叫回調函數，應該是函數引用，但也可用函數名代替。其意義就是在指定時間後調用那個回調函數，而不影響現在 vim 對用戶的正常響應。還可以指定可選的第三參數，表示重複回調若干次，默認就只回調一次，然後自動關閉定時器。該函數有返回值，表示定時器 ID，在 vim 內部就用該 ID 標記這個定時器。回調函數一般是自定義函數，必須接收一個表示定時器 ID 的參數。不過在這個簡單示例中，我們忽略未用到這個定時器 ID 參數。定時器相關函數的詳細用法請參考 ':help timer-functions'。

由此可見，異步編程的基本思路是將原來在一個函數內的工作（一般是較費時的工作），多拆出一個回調函數，用來在工作完成時處理 "後事"，關鍵也就是回調函數的編寫。在這個例子中，我們用定時器來 "模擬" 了一件慢工作，當然定時器本身也另有用途場景。

定時器可以明確指定延時幾秒，不過在實際的慢工作（外部命令）中，需要多長時間完成工作是不確定的。這就需要另外的機制，根據其他條件來調用回調函數。這就是下一節准備講的 "任務"，原文檔術語叫 job 的話題了。

8.2 使用异步任務

注意：本節所介紹的功能要求 vim 編譯包括 +job 特性。

簡單任務體驗

前文說到，Vim 的異步任務主要是針對外部命令的。那我們就先以最簡單最常見的系統命令 ls 爲例，其功能是列出當前目錄下的文件，若在 Windows 操作系統下或可用 dir 命令代替。

首先請在 shell 中進入一個非空目錄，便於實踐，並在 shell 中執行如下命令：

```
$ ls
```

然後啓動 vim 中，在 vim 命令行中執行如下命令：

```
:!ls
```

體驗一下 vim 直接執行外部命令的現象。與在 shell 中執行幾乎是一樣的，只是將輸出打印到終端，供用戶交互時查看。然而在用腳本編程中，我們一般希望將外部命令的輸出保存到某個變量，便於後續控制與利用。如此可用 system() 函數：

```
: let g:dir_list = system('ls')
: echo g:dir_list
```

當然一般而言，ls 命令執行得足夠快，在 VimL 腳本中能很快捕獲到其輸出。不過我們暫時忽略外部命令的速率，再來看來如何用異步任務完成類似的任務。

```
1  function! OnWorking(job, msg)
2      echomsg 'well work doing:' . a:msg
3      let g:dir_list .= a:msg . "\n"
4  endfunction
5
6  function! DoneWork(job)
7      echomsg 'well work done:'
8      echomsg g:dir_list
9  endfunction
10
11 function! StartWork()
12     let g:dir_list = ''
13     let l:option = {'callback': 'OnWorking',
14                     'close_cb': 'DoneWork'}
15     let g:job_ls = job_start('ls', l:option)
16 endfunction
```

在這個示例中，函數中直接使用全局 'g:' 變量，並非良好編程規範，這裏僅作說明目的，便於在命令中測試觀察。在命令中輸入 ':call StarWork()' 運行示例。

內置函數 job_start() 用於開啓一個異步命令。其第一參數就如同 system() 函數的參

數，指定要運行的外部系統命令。第二個可選參數是個有諸多鍵的字典，用於配置或控制該任務的行爲。其中最重要的參數就是設置回調函數，在該示例中指定了兩個回調函數。一個是 OnWrking() 在工作進行時調用，每當所執行的任務有輸出時就會被調用，輸出會通過第二參數傳入回調函數；另一個是 DoneWork() ，在工作完成時調用。當然應該知道，這兩個函數名是我們任意自定義的，名字不重要，關鍵的魔法是鍵名，callback 與 close_cb 標識了對應的函數（引用）在適當的機會被調用。

這兩個回調函數爲求簡單，忽略了第一個作爲任務標識的參數，並且仍利用全局變量 g:dir_list ，在工作進行時將外部輸出收集（串接）起來，最後在工作完成時一次性地將其完整地用 VimL 打印出來，或爲其他更有價值的利用。這裏用 echomsg 而不是 echo 命令是爲了能在隨後（通過‘:message’）查看消息歷史記錄。不過要注意，雖然 g:dir_list 在串接時添加了回車符變成多行文本，但 echomsg 仍將其當作一行輸出，於是回車符會被其他可打印符號（@）代替。可手動執行‘:echo g:dir_list’再確認它是多行文本，或用 split() 函數將其分隔到列表中。

另一點要注意的是，並非每次 job_start() 啓動任務都得註冊這兩個回調，根據實際工作任務情況可在其中一個（或更多）回調函數中處理感興趣的信息。甚至如果只是想讓某個外部命令在後臺默默運行，不關心任何反饋的話，也可以不註冊任何回調函數。譬如在後臺用 ctags 更新索引文件。只不過提供回調的話，會使異步任務更有交互感與確認感，讓用戶知道後臺命令確實在執行了。

job 選項及其他相關函數

job_start() 的第二參數支持相當多的選項，詳情請見‘:help job-options’，這裏擇其要點解釋相關概念。幫助主題中所列的選項，不僅給這個 job_start() 函數使用，也供更通用的底層“通道” ch_open() 利用，後者在下一節繼續介紹。現在只需理解，任務（job）是通道（channel）的一種特例或具體應用。

任務採用管道（pipe）將外部命令與 vim 聯接起來，那就涉及標准輸入、標准輸出與標准錯誤輸出這三套件，在其間的消息傳遞都採用所謂 NL 模式，可以理解爲輸入輸出都按回車分行的字符串。如果某個輸出/輸入端是有格式的消息字符串（如 json），則可通過 in_mode out_mode err_mode 分別設定。不過在大多數情況下，使用默認的 NL 模式就適合，且理解更爲自然，當然這實際上取決於所調用的外部命令的需求。

本節開始的示例所謂 callback 回調，其實能同時捕獲標准輸出與標准錯誤輸出，也就是假設外部命令直接在 shell 中執行會打印到屏幕終端的所有可見信息。如果想更精細地區分兩者，那就使用 out_cb 與 err_cb 這兩種回調，各司其職。

與 close_cb 類似的回調，還有個 exit_cb 回調。從字面上理解，前者是任務關閉時調用，後者是退出時被調用。exit_cb 回調函數比 close_cb 可多接收一個參數，表示任務的狀態。

任務管道使用輸入輸出還可以重定向到文件，或在 vim 中打開的一個 buffer，使用 in_io out_io err_io 及相關的選項設置。如果捕獲輸出不是最終目的，就可避免在回調函數中將輸出保存至 VimL 變量中，直接設置 out_io 輸出至 buffer 中呈現更爲直觀。

例如，有這麼一個命令 tail -f 可用於監控持續增長的日誌文件。如果要從 vim 調用它，在支持異步特性之前，若用 system() 函數，它永遠不會返回，那便無用。然而用 job_start() 啓動它，再將 out_io 設置爲一個 buffer ，就可以達到目的，直接在 vim 中查看增長中的日誌。

當然了，還是使用回調函數的工作流更常見，畢竟編程控制上更靈活。如果最終仍想在某個 vim buffer 中展示輸出，quickfix 或 localist 或許也是個更好的選擇。譬如異步執行 grep （或其他更佳的搜索工具），將結果放在 quickfix 中也適於跳轉。

job_start() 也是有返回值的，返回一個標記，代表這個啓動的任務，能傳遞給其他幾個任務相關的函數，以指明操作哪個任務。job_stop() 停止指定任務，如果啓動的外部命令是設計爲死循環永不終止的，也許在 VimL 中就有必要用該函數顯式終止任務了。job_status() 用於查詢一個任務的狀態：fail 表示任務根本就沒成功啓動；run 表示任務正常進行中；dead 表示任務跑完了。job_info() 則可查詢有關任務更詳細的信息。

一般來說，任務的選項是要在啓動時設置，但也有些選項可以在啓動之後，還處於 run 狀態時，使用 job_setoptions() 補充選項。這運用場景就有些受限了。最後，還有個函數 job_getchannel() 用於獲得任務底層的通道。

通用異步插件 asyncrun

job 選項與細節繁多，除了幫助文檔，另一個絕好的學習方式是參考優秀插件的實現與運用。這裏隆重推薦asyncrun，出自國人網名 “韋一笑” 大神。如果只是使用，它已經封裝得很好了，直接使用 AsyncRun 命令即可。如果是想學習異步編程，則該插件也足夠輕量，只有一個單文件，也非常適合參考學習。

比如，瀏覽大概後直接搜索 job_start 看它是如何啓動異步任務的，摘錄關鍵代碼如下：

```
let l:options = {}
let l:options['callback']=function('s:AsyncRun_Job_OnCallback')
let l:options['close_cb']=function('s:AsyncRun_Job_OnClose')
let l:options['exit_cb'] = function('s:AsyncRun_Job_OnExit')
let l:options['out_io'] = 'pipe'
let l:options['err_io'] = 'out'
let l:options['in_io'] = 'null'
let l:options['out_mode'] = 'nl'
let l:options['err_mode'] = 'nl'
let l:options['stoponexit'] = 'term'
if g:asyncrun_stop != ''
    let l:options['stoponexit'] = g:asyncrun_stop
endif
if s:async_info.range > 0
```

```
15      let l:options['in_io'] = 'buffer'
16      let l:options['in_mode'] = 'nl'
17      let l:options['in_buf'] = s:async_info.range_buf
18      let l:options['in_top'] = s:async_info.range_top
19      let l:options['in_bot'] = s:async_info.range_bot
20  endif
21  let s:async_job = job_start(l:args, l:options)
22  let l:success = (job_status(s:async_job) != 'fail')? 1 : 0
```

可見，它首先是詳細構建選項字典，關鍵的回調函數顯然是引用腳本私有函數的。注意在那個條件分支中設定 in_io 標准輸入選項，那是在指定選區時運行 ':<,'>AysncRun' 時傳入的，把當前 buffer 選定的行供給任務的標准輸入。在 job_start() 之後，再立即調用 job_status() ，可判斷任務是否成功啓動過。

然後按圖索驥，跟蹤賦給選項的變量從哪裏來，回調函數處理又到哪裏去（也正是添加到 quickfix 窗口中）。除此之後，就如常規的 VimL 編程了。

你可以利用該插件體驗一下在 vim 中直接執行 make 編譯或 grep 搜索：

```
: AsyncRun make
: AsyncRun grep ...
```

對比體驗一下在 vim7 之前沒有異步支持時只能用類似如下的命令：

```
:! make
:! grep ...
```

小結

異步任務只是 vim8 開始引入的新機制，爲解決某些問題尤其是調用外部耗時命令時提供另一種編程模式。要真正利用好異步機制，自然還取決於整體的 VimL 編程技術，比如如何有效地管理變量與函數這種基礎水平。不過，如果有在其他語言編寫過異步回調的經驗，改用 VimL 編寫異步任務也是類似的思想，就更容易上手些。

8.3 使用通道控制任務

通道的概念

Vim 手冊上的術語 channel 直譯爲通道，比起任務 job 聽來更爲抽象。上一節介紹的任務，直觀想起來，即使不是瞬時能完成的 “慢” 命令，也是一項 “短命” 的命令，可以期望它完成，也就完成了任務。

顯然，我們可以用 job_start() 同時開啓幾個異步命令，但是如果企圖通過這方式開啓一組貌似相關的任務，可能達不到目的。因爲開啓的不同任務相互之間是獨立的，各自獨立在後臺運行。比如，連續開啓以下兩個命令：

```
: call job_start('cd ~/.vim/')
: call job_start('ls')
```

這兩條語句寫在一起，並不能（或許有人想當然那樣）進入目標目錄後列出文件。第一條語句開啓一個後臺命令 cd 進入目錄，但是什麼也沒干就完成了；第二條語句開啓另一個獨立的後臺命令 ls 仍然是列出當前目錄的文件。

不過這個需求在 vim 中是有解決辦法的，想想在 vim8.1 中隨着異步特性增加的內置終端的功能，顯然是可以通過開啓內置終端，在此內置終端中輸入 cd ls 命令列出目標目錄下的所有文件：

```
: terminal
$ cd ~/.vim
$ ls
```

既然可以在 vim 中列出一串內容，可想而知也有他法將列出的內容捕獲到 VimL 變量中，再進行想要的程序邏輯加工。

‘:terminal’ 命令其實有個默認參數，就是異步開啓一個交互 shell 進程（如 bash），只不過這個任務與上一節介紹的異步任務有所不同，特殊在於它是不會主動結束的，相當於一個無限死循環等待用戶輸入，再解釋執行（shell 命令）給出迴應。那麼 vim 與後臺異步開啓的這個 shell 進程（任務），肯定是該有個東西連着，以促成相互之前的通訊，這個東西就叫做 “通道”，也就是 channel 。

通道的一端自然是連着 vim ，另一端一般連着的是能長期運行的服務程序。上一節介紹的異步任務，也是有個通道連着外部命令的，如此 vim 才能知道外部命令有輸出，什麼時間結束，才能在適當時機調用回調函數。只不過那外部命令自然結束後，通道也就斷了。所以最好反過來理解，通道纔是底層更通用的機制，任務是一種短平快的特殊通道。

Vim 的在線文檔 ‘:help channel’ 專門有個文檔來描敘通道（及任務）的使用細節，並且在一開始還有個用 python 寫的簡單服務程序，用於演示 vim 的通道聯連與交互。對 python 有親切感的讀者，可以好好跟一下這個演示示例。從這麼簡單樸素的服務開始，通道可以實現複雜如內置終端這樣的標誌性功能。雖然我們學 VimL ，不求一下子就能寫那麼複雜的高級功能，但理解通道的機制，掌握通道的用法，也就能大大擴展 VimL 編程的效能，滿足在舊版本所無法實現的需求。

開啓通道與模式選項

要開啓一個通道，使用 ch_open() 函數，我們將其函數 “原型” 與前面兩節介紹的定時器、任務的啓動函數放在一起對照來看：

- 定時：timer_start({time}, {callback} [, {options}])

- 任務：job_start({command} [, {options}])
- 通道：ch_open({address} [, {options}])

定時器的第一參數是時間，因爲它是將在確定的時間內執行工作，同時定時器要有效用，也必須在第二參數處提供回調函數，以表示到那時執行具體的動作。而任務，是無法提前得知執行外部命令需要多少（毫秒）時間的。所以啓動任務的第一參數，就是外部命令，有時這就夠了，只要讓它在後臺默默完成即可；之後的選項是可選的，而且對於複雜任務，也可能需要幾種不同時機的回調，故而全部打包在一個較大的選項字典中，令使用接口簡單清晰。

至於通道，它更抽象在於，它其實不是針對具體命令的，而是針對某個“地址”，就如 socket 編程範疇的“主機: 端口”的地址概念。Vim 的通道就是可以聯接到這樣的地址，與其另一端的服務進行通訊，至於另一端的服務是由什麼命令、由什麼語言寫的程序，這不需要關心，也不影響。

在通道的選項集中，除了同樣重要的回調函數外，還有個更基礎的模式選項須得關注，就是叫 mode 的。模式規定了 Vim 與另一端的程序通訊時的消息格式，粗略地講，可直觀地理解爲傳輸、讀寫的字符串格式。共支持四種模式，上一節介紹的由 job_start() 啓動的任務默認就是使用 NL 模式，意爲 newline ，換行符分隔每個消息（字符串）。這裏使用 ch_open() 開啓的通道默認使用 json 格式。json 是目前互聯網上很流行的格式，vim 現在也內置了 json 的解析，所以使用方便靈活。

另外兩種模式叫做 js 與 raw 。js 模式是與 json 類似的、以 javascript 風格的格式，文檔上說效率比 json 好些。因爲 js 編碼解碼沒那麼多雙引號，以及可省略空值。raw 是原始格式之意，也就是沒任何特殊格式，vim 對此無法作出任何假設與預處理，全要由用戶在回調函數中處理。

至於在具體的 VimL 編程實踐中，該使用哪種模式的通道，這取決於要連接的另一端的程序如何提供服務了。如果能提供 json 或 js 最好，要不 NL 模式簡單，如果連換行符也不一定能保證，那就只能用 raw 了。如果另一端的程序也是由自己開發，那掌握權就更大了，如果簡單的可以用 NL 模式，複雜的服務就推薦 json 了。

模式之所以重要，是因爲它深刻影響了回調函數的寫法。比如 vim 從通道中每次收到消息，就會調用 callback 選項指定的函數（引用），並向它傳遞兩個參數；故回調函數一般是形如這樣的：

```
1 function! Callback_Handler(channel, msg)
2     echo 'Received: ' . a:msg
3 endfunction
```

其中第一參數 a:channel 是通道 ID ，就是 ch_open() 的返回值，代表某個特定的通道（顯然可以同時運行多個通道）。第二參數 a:msg 所謂的消息，就與通道模式有關了。如果是 json 或 js 模式，雖然 vim 收到的消息初始也是字符串，但 vim 自動給你解碼了，於是 a:msg 就轉換爲 VimL 數據類型了，比如可能是富有嵌套的字典與列表結構。如果是 NL 模式，則是去除換行符的字符串；當然如果是 raw 模式，那就是最原始的消息了，可能有

的換行符也得用戶在回調中注意處理。

通道交互

與任務不同的是，通道僅僅由 ch_open() 開啓是不夠的。那只是建立了連接，告訴你已經准備好可以與另一端的程序服務協同工作了。但一般它不會自動做具體的工作，需要讓 vim 與彼端的服務互通消息，告訴對方我想干什麼，請求對方幫忙完成，並（異步或同步地）等待迴應。雖然有些服務可以主動向 vim 發一些消息，讓 vim 自動處理，但畢竟有限，你也不能放任外部程序不加引導控制地影響 vim 是不。所以，有來有往的消息傳遞，纔是通道常規操作，也是其功能強大所在。

互通消息的方式，也與通道模式有關。

向 json 或 js 模式的通道（彼端）發消息，推薦如下三種方式之一：

1. call ch_sendexpr(channel, expr)
2. call ch_sendexpr(channel, expr, 'callback': Handler)
3. let response = ch_evalexpr(channel, expr)

注意前兩種寫法，直接用 ':call' 命令調用函數，忽略函數返回值。它單純地發送消息，異步等待迴應；當之後某個時刻收到響應後，就調用通道的回調函數。但是如第二種用法，在發送消息時提供額外選項，單獨指定這條消息的回調函數。

於是就要有一種機制來區分哪條消息，vim 在發送消息時實際上發送 [{number},{expr}]，即在消息之前附加一個編號，組成一個二元列表。該編號是 vim 內部處理的，一般是遞增保證唯一，{expr} 纔是由程序員指定的 VimL 有效數值（或數據結構），並再由 vim 編碼成 json 字符串，或 js 風格的類似字符串。通道彼端接收到這樣的消息，將 json 字符串解碼，經其內部處理後，再由通道發還給 vim ，並且也是由編號、消息體組成的二元列表 [{number},{response}]。在同一請求——迴應中，編號是相同的，vim 據此就能分發到對應的回調函數，傳入的第二參數也就是 {response} ，不包含編號的消息主體。當然，按第一種寫法未指定回調地發送消息，收到響應時就會默認分到在 ch_open() 中指定的回調函數中。

至於第三種寫法，一般要用 ':let' 命令獲取 ch_evalexpr() 的返回值。這是同步等待，就如 system() 函數捕獲輸出一樣。同步雖然可能阻塞，但優點是程序邏輯簡單，不必管回調函數那麼繞。在通道已經建立的情況下，如果另一端的服務程序也運行在本地機器，ch_evalexpr() 可能比 system() 快些。因此，如果預期將要請求執行的操作並不太複雜時，可儘量用這種同步消息組織編程。另外，通道也有個超時選項，不致於讓 vim 陷入無限等待的惡劣情況。在超時或出錯情況下，ch_evalexpr() 返回空字符中，否則返回的也是已解碼的 VimL 數據，如同 ch_sendexpr() 收到迴應時傳給回調函數的消息主體。

對於 NL 或 raw 模式，無法使用上面這兩個函數交互，應該使用另外兩個對應的函數：

1. call ch_sendraw(channel, {string})
2. call ch_sendraw(channel, {string}, {'callback': 'MyHandler'})
3. let response = ch_evalraw(channel, {string})

其中第二參數必須是字符串，而不能是其他複雜的 VimL 數據結構，並且可能需要手動添加末尾換行符（視通道彼端程序需求而論）。

json 與 js 模式的通道也能用 ch_sendraw() 與 ch_evalraw() ，不過需要事先調用 json_encode() 將要發送的 VimL 數據轉換（編碼）爲 json 字符串再傳給這倆函數；然後在收到響應時，又要將響應消息用 json_decode() 解碼以獲得方便可用 VimL 數據。

因此，所謂通道的四種模式，是指通道的 vim 這端如何處理消息的方式，vim 能在多大程度上自動處理消息的區別上。至於通道另一端如何處理消息，那就不是 vim 所能管的事了，是那邊的程序設計話題。也許那邊的程序也有個網絡框架自動將 json 解碼轉化爲目標語言的內部數據，或者要需要手動調用 json 庫的相關函數，再或者是簡單粗暴地自己解析 json 字符串……那都與 vim 這邊無關了，它們之間只是達到一個協議，需要傳輸一個兩邊都能正確解析的字符串（消息字節）就可以了。

此外還得辨別另一個概念，通道的這四種解析模式，與通道的兩種通訊模式又不是同一層次的東西。後者指的是 socket 或管道（pipe），是與操作系統進程間通訊的更底層的概念，前者 json 或 NL 卻是 VimL 應用層面的模式。上一節介紹的任務，由 job_start() 啓動的，使用的是管道，重定向了標准輸入輸出與錯誤；這一節介紹的通道，由 ch_open() 開啓的，使用的是 socket ，綁定到了特定的端口地址。然後，在 vim 中，任務的管道，也視爲一種特殊通道。

通道示例：自制簡易的 vim-終端

本節的最後，打算介紹一個網友寫的模擬終端插件。

這應該是在 vim8.1 暫時未推出內置終端，但先提供了 +job 與 +channel 時寫的插件，目的在於直接在 vim 中模擬終端，執行 shell 命令。雖然沒有後來 vim 內置終端那麼功能強大，但也頗有自己的特色。關鍵是還比較輕量，代碼量不多，可用之學習一下如何使用 vim 任務與通道的異步功能。借鑑、閱讀源碼也正是學習任何語言編程的絕好法門。

首先應該瞭解，作爲發佈在 github 上的插件，或多或少都會追求某些通用性，於是在插件中就不可避免涉及許多配置，比如全局變量的判斷與設置。就像這個插件，它想同時用於 vim 與 nvim ，兩者在異步功能上可能提供了略有不同的內置函數接口，然而還想兼容 vim7 低版本下沒異步功能時退回使用 system() 代替。

拋開這些“干擾”信息，直擊關鍵代碼，看看如何使用 vim 的異步功能吧。從功能說明入手，它主要是提供了:ZFTerminal 命令，在源碼中尋找該命令定義，獲知它所調用的私有函數 s:zfterminal：

```
1 command! -nargs=* -complete=file
2     / ZFTerminal :call s:zfterminal(<q-args>)
3 function! s:zfterminal(...)
4     let arg = get(a:, 1, '')
5     " ... (省略)
6     let needSend=!empty(arg)
```

```
    if exists('b:job')
        let needSend=1
    else
        call s:updateConfig()
        let job = s:job_start(s:shell)
        let handle = s:job_getchannel(job)
        call s:initialize()
        let b:job = job
        let b:handle = handle
        if exists('g:ZFVimTerminal_onStart')
            / && g:ZFVimTerminal_onStart!=''
            execute 'g:ZFVimTerminal_onStart(' . b:job . ',
            / ' . b:handle . ')'
        endif
    endif
    if needSend
        silent! call s:ch_sendraw(b:handle, arg . "\n")
    endif
    " ... (省略)
endfunction
```

它這裏的思路是將開啓的任務保存在 b:job 中。這很有必要，因爲隨後的回調函數都要用到任務 ID（功通道 ID)。它不能保存在函數中的局部變量中，否則離開函數作用域就不可引用該 ID 了，也不宜污染全局變量。於是腳本級的 s: 變量合適；如果異步任務始終與某個 buffer 關聯，則保存在 b: 作用域更清晰，且容易支持多個任務並行。ZFTerminal 正是將一個普通 buffer 當作 shell 前端來用，因而保存爲 b:job。

如果在執行命令時，任務不存在，就用 job_start() 開始一個任務，否則就向與任務關聯的通道用 ch_sendraw() 發送消息。它爲這兩個函數再作了一個淺層包裝（主要爲兼容代碼考量及定義一些默認選項)。job_start() 它是這樣開啓的：

```
function! s:job_start(command)
    " ...
    return job_start(a:command, {
    \   'exit_cb' : 'ZFVimTerminal#exitcb',
    \   'out_cb' : 'ZFVimTerminal#outcb_vim',
    \   'err_cb' : 'ZFVimTerminal#outcb_vim',
    \   'stoponexit' : 'kill',
    \   'mode': 'raw',
    \ })
```

```
10 endfunction
```

在這裏它指定了幾個回調函數，並將通道模式設爲 raw 。所以在後續‘:ZFTerminal’命令中就用 ch_sendraw() 發送消息了。注意發送消息需要通道 ID 參數，使用 job_getchannel() 函數可以獲取相任務關聯的通道，並且也保存在 ‘b:’ 作用域內。至於回調函數，請自行結合所實現的功能跟蹤，此不再贅述。

8.4 使用配置內置終端

使用異步的兩個方面

本章討論的是 vim 的異步特性，其實這包含兩個方面。其一如何利用 VimL 編程控制異步任務，（寫插件）實現特定的功能。前三節都是圍繞這個話題的，從簡單到複雜介紹了 vim 提供的三種異步機制，定時器、任務與通道。那可能有點抽象或晦澀，需要與具體的插件功能結合起來才更好理解，但是基於本書的定位，也不便介紹與解讀太複雜的插件。

其二是如何更好地使用 vim 新版本自身提供的異步功能，典型的就是內置終端。作爲普通用戶，相對於開發，運用可能是更簡單有趣的。本節就是打算跳出複雜的異步編程的曲折過程，調劑一下，重新回到簡單常規的 VimL 調教與定製內置終端，使之更符合個性習慣，成爲日常使用的利器。

當然，這依然是引導性質或經驗之談，詳細文檔請看‘:help terminal’。

內置終端的啓動

```
: terminal
: terminal bash
: terminal python
```

用‘:terminal’命令開啓內置終端。其實廣義來講，它可以接受外部命令參數，在內置終端中運行任意的外部命令，譬如打開一個 python 解釋器。默認無參數時就執行 &shell 指定的程序，比如 bash 。

不過一般地，我們提到內置終端，就是指狹義上的在 vim 裏面運行一個 shell 。它會橫向分裂一個半屏窗口，在這個特殊的窗口就幾乎與外面運行的 shell 一樣的操作與功能，包括比如.bashrc 的 shell 配置。

‘:terminal’除了可以指定外部命令參數外，還可以接受許多選項，控制諸如內置終端的窗口大小、位置等各種選項。你可以將自己的偏好啓動選項封裝起來，自定義一個函數、命令或快捷鍵。

此外，除了 vim 命令，還有個 vim 函數 term_start() 用於在編程邏輯中啓動一個內置終端，用法就如 job_start() 一樣，給予靈活控制，按需啓動終端。

終端模式的快捷鍵映射

在打開的內置終端窗口，爲了能像外部 shell 那樣使用 shell 本身的快捷鍵，Vim 禁用了絕大部分快捷鍵。雖然在內置終端窗口中可以鍵入 shell 命令，但那不是 vim 的插入模式也不是命令行模式，所以 imap 與 cmap 都不生效，當然更不可能是普通模式了。事實上，Vim 爲此專門新定義了一種特殊模式，叫“終端任務”（Terminal-Job）模式，不妨簡稱終端模塊。如果要爲終端模式自定義快捷鍵，應該用 tmap 系列命令。

不過在動手之前，還是要了解 vim 已經保留了一個特殊鍵用來切換回 vim 的普通模式；而且由於前敘原因，也僅保留了一個鍵。這個鍵由選項 &termwinkey 給出，默認也是 <C-W> ，因爲它的本意正是如何使用‘:wincmd’切出終端窗口。於是 <C-W> 引導的快捷鍵在終端窗口與普通窗口保持一致的含義，並且附帶兩個擴展：

- <C-W>w 切到下一窗口，<C-W>W 切到上一窗口，<C-W>p 切到之前所在窗口

……

- <C-W>n 或 <C-W><C-N> 終端窗口切到普通模式（可以用 hkl 移動了）
- <C-W>: 從終端窗口進入 vim 命令行，（否則按冒號只是在 shell 提示符後輸入冒號呢）

如果不喜歡 <C-W> 這個引導鍵——比如說因爲 <C-W> 在 shell 中是刪除前面一個詞的快捷鍵，故想將 <C-W> 鍵傳給 shell ——那麼可以設置 &termwinkey 更換。但一般不建議修改，保持 Vim 內換窗口操作一致性較爲重要，況且換任何鍵都可能會與 shell 衝突，總之是需要權衡。

從終端的任務模式回到普通模式略爲麻煩，要按 <C-W><C-N> （在已經按下 <C-W> 的情況下，<C-N> 多按或少按那個 ctrl 鍵差別不大了）。爲什麼不保留 <Esc> 鍵回到普通模式呢？大概是 vim 想兼容更多的終端，有些終端用 <Ecs> 作爲轉義符。就個人使用經驗而言，在 shell 中會經常用到 <Ecs>. （接一個點）快捷鍵輸入上一條命令最後一個詞。不過權衡之下，可以重定義 <Esc> 回到普通模式：

```
tnoremap <Esc> <C-\><C-N>
tnoremap <C-W>b <C-\><C-N><C-B>
tnoremap <C-W>n <C-W>:tabnext<CR>
tnoremap <C-W>N <C-W>:tabNext<CR>
tnoremap <C-W>1 <C-W>:1tabNext<CR>
tnoremap <C-W>2 <C-W>:2tabNext<CR>
...
```

除了 <Esc> 鍵外，我還定義了其他幾個快捷鍵。比如使用終端時需要經常上翻查看結果，就在 <C-W> 引導鍵後加個 b ，回到普通模式的同時上翻一頁。然後我自己用 tabpage（標籤頁）比較多，所以也用 <C-W> 加數字切到特定的標籤頁中。當然，明白了 tnoremap 之後，就能像 nnoremap 一樣按自己習慣重定義快捷鍵了。

另外，按特定方式啓動終端也可以自定義方便的快捷鍵，不過我推薦另一種思路，短命令，例如：

```
command! -nargs=* TT tab terminal <args>
command! -nargs=* TV vertical terminal <args>
```

這意思是用‘:TT’命令在另一個標籤頁打開終端，用:TV 按縱向分割窗口打開終端。可以將其想象爲 <mapleader> 是‘:’，而且冒號本來就要按下 shift 鍵，再接一兩個大寫字母也順手，只不過最後還要多按 <CR> 回車確認執行命令。然而這另有個好處是還可以隨時增加其他命令行參數（傳給‘:terminal’），這種靈活性是普通模式下的快捷鍵不能達成的。因此，“短命令”適合於替代那些“次常用”的快捷鍵，畢竟鍵盤佈局的快捷鍵資源以及個人的記憶習慣是有限的。

既然內置終端的啓動方式可以定製，那麼就想如何能在啓動終端時才自動定義那些 tmap 快捷鍵呢？畢竟 tmap 在平時是用不上，也未必是每次打開 vim 都會用到內置終端，將 tmap （及其他與終端相關的設置）直接寫在全局 vimrc 有點“浪費”。vim 顯然也想到了這個需求，很貼心地增加了一個自動命令事件，TerminalOpen 就會在打開內置終端窗口時觸發，於是可將如下事件寫在某個合適的事件組（augroup）中：

```
autocmd! TerminalOpen * call OnTermialOpen()
```

將你想要定製內置終端的代碼都寫在 OnTerminalOpen() 函數中，當然使用 # 形式的自動加載函數會更好。

內置終端與 vim 交互

所謂交互，自然是分兩方面的。其中從內置終端（的任務中）向 vim 發起交互的需求，可能來自一個有趣的“哲學”問題：可不可以在內置終端中輸入 $ vim file 再啓一個 vim 編輯文件呢。那自然是可以的，但在使用中那顯得有點愚蠢，不夠優雅。於是，就需要一個機制，從內置終端中向開啓它的“宿主”vim 發送消息，令其打開某個文件。

於是 vim 就有了這麼個約定（據說來自 emacs），在內置終端運行的程序，只要向標准輸出打印如下序列：

```
<Esc>]51;["drop", "filenmae"]<07>
```

實際上就會將 [“drop” ， “filenmae”] 傳遞給宿主 vim ， 然後 vim 就知道將該消息解釋爲執行‘:drop filename’命令。‘:drop’命令其實與‘:edit’命令類似，就是打開一個文件，只不過如果文件已被打開，就會跳到相應的目標窗口。:drop 命令也就是隨內置終端版本一起增加的，可見它的原意就是想解決這個痛點。

Esc 字符是終端的轉義符，在 VimL 中固然可以用 <Esc> 表示，但在其他語言（如 C 語言）中，則一般用 \e 表示，或直接用其 ASCII 碼（\x1B 或 \033 即十進制的 27）表示。

例如，可以在 ~/bin/ 目錄下寫個簡單的 drop.sh 腳本：

```
1 #! /bin/bash
2 echo -e "\e]51;[\"drop\", \"$1\"]\x07"
```

注意傳給 vim 的消息要求是 json 模式（見前一節的通道模式），drop 與文件名參數須按 json 標准用雙引號括起。在多數語言或腳本中如果用雙引號括起整個序列字符串，就得將裏面的 json 字符串的雙引號用 \" 轉義。可以用其他任何語言寫這個 drop 腳本，例如等效的 perl 腳本（dorp.pl）可以如下：

```
1 #! /usr/bin/env perl
2 my $filename = shift;
3 print qq{\x1B]51;["drop", "$filename"]\x07};
```

然後爲了使用習慣，可以再在 ~/bin/ 中建個 drop 軟鏈接，指向實用的 drop 腳本，如：

```
$ chmod +x drop.pl
$ ln -s drop.pl drop
```

如果 /bin 在環境變量 PATH 中，則在 vim 的內置終端中，執行如下命令：

```
vim-shell $ drop file
```

就能在宿主 vim 中用 ‘:drop’ 打開相應的文件。不過這還有個問題。我們在 shell 中給任何命令輸入文件名參數，一般都是當前目錄下的文件名。但是 vim 內置終端的當前目錄，很可能與宿主 vim 的當前目錄並不相同，於是 drop 命令可能會失效，所以在傳遞消息中應該使用絕對路徑，以保證能找到正確的文件。爲此，可將 ~/bin/drop.pl 改爲如下：

```
1 #! /usr/bin/env perl
2 use Cwd 'abs_path';
3 my $filename = shift or die "usage: dorp filename";
4 my $filepath = abs_path($filename);
5 exec "vim $filepath" unless $ENV{VIM};
6 print qq{\x1B]51;["drop", "$filepath"]\x07};
```

主要改動是利用語言的相關模塊獲取文件絕對路徑，並稍微保護判斷下是否是否輸入了文件名參數。另一個改動是倒數第二行 exec ... unless 語句。只有在 vim 的內置終端纔會向宿主 vim 發 drop 消息，如果是從外部普通 shell 使用該腳本，那就會改爲啓動 vim（進程覆蓋當前進程）打開命令行指定的文件，而最後一行再也沒機會執行了。從 vim 中啓動的內置終端會繼承 vim 進程的環境變量，至少它會有 $VIM 這個環境變量（可以用:echo $VIM 查看），據此可以判斷是內置終端還是外部終端。

當然，如果你熟悉 python ，用 python 寫個 drop.py 也是容易的。

在 <Esc>51;[msg]<07> 轉義序列中向 vim 傳遞的消息，除了支持 ‘:dorp’ 命令，還支持 ‘:call’ 命令調用特殊的以 Tapi_ 開頭的自定義函數（限定函數名規範是爲安全起見）。消息形如 [“call” , “Tapi_funcname” , [argument-list]] 。自定義函數約定接受兩個參數，與內置終端窗口關聯的 buffer 編號，以及一個參數，所以如果業務邏輯需要多個參數，就只能將它們打包在一個列表或字典類型的變量，當作一個參數傳入。Vim 開放這麼個接口提供靈活擴展的可能，具體能做什麼那當然是用戶的實現了。

vim 與內置終端交互

交互的另一方面，是 vim 向內置終端發消息。

顯示，內置終端也是個任務，有着底層的通道，所以始終可以嘗試使用上節介紹的 ch_sendraw() 等函數。然而對於內置終端，沒必要使用底層的函數，vim 提供更高層函數 term_sendkeys() 直接向內置終端發送一個字符串，效果如同在終端提示符下手動鍵入。注意該函數與 feedkeys() 的區別，後者是相當於向 vim 鍵入字符串，會被 vim 截獲，並受 tmap 映射影響；而前者是直接向內置終端鍵入，不受 tmap 影響。

試驗一下，在打開內置終端的窗口中，使用 <C-W>: 進入命令行，輸入：

```
: call feedkeys('ls')
```

回車執行後，會在內置終端的提示符之後顯示 ls 這兩個字符，那就是相當於用戶通過 vim 界面向內置終端敲了兩個字符，但還沒敲回車真正發送給內置終端運行。你可以繼續編輯這個命令，比如使用退格鍵刪除之，或在其後增加選項 -l ，然後再按一次回車，內置終端才能響應執行這個 ls 命令。然後，再 <C-W>: 試試輸入：

```
: call term_sendkeys('', 'ls')
```

發現效果似乎還是一樣，ls 這兩字符停在內置終端提示符之後等待執行。需要將回車鍵與合在這兩個函數的參數中，纔是通知內置終端立即執行：

```
: call feedkeys('ls' . "\<CR>")
: call term_sendkeys('', 'ls' . "\<CR>")
```

注意，回車鍵 <CR> 需要雙引號轉義。並且 term_sendkeys() 函數要求第一個參數是指定內置終端的 buffer 編號，空值表示當前內置終端。從用戶角度看，如果不涉及（少量的）被 tmap 映射的鍵序列，用這兩個函數的效果基本相同，但爲了安全起見以及語義明確，向內置終端發消息時，最好用 term_sendkeys() 函數。

在上一節介紹的 ZFVimTerminal 插件有個特性，是從 vim 的命令行中向模擬終端發送命令。我們也可以借鑑這個思路，實現從 vim 命令行中向內置終端發送命令。當然了，從內置終端窗口本身再用 <C-W>: 進入命令行輸命令就有點多此一舉了，反而麻煩。所以需求應該是從任何一個普通 buffer 窗口，按: 後在命令行向內置終端發送命令，避免需要跳到內置終端的麻煩；當內置終端不存在時，顯然應該打開一個新的內置終端。

爲此，可以封裝一個函數，並定義命令調用該函數，大致如下：

```
1 command! -nargs=* -bang TC
2  / call useterm#shell#SendShellCmd(<bang>0, <q-args>)
3 command! -nargs=* TCD
4  / call useterm#shell#SendShellCmd(0, 'cd ' . expand('%:p:h'))
5
6 function! useterm#shell#SendShellCmd(bang, cmd) abort
7     " save current window
```

```
 8      if a:bang
 9          let l:tab = tabpagenr()
10          let l:win = winnr()
11      endif
12
13      let l:found = useterm#shell#GotoTermWin(&shell)
14      if empty(l:found)
15          :terminal
16      endif
17      if !empty(a:cmd)
18          call term_sendkeys('', a:cmd . "\<CR>")
19          " into insert mode to force redraw terminal window
20          normal! i
21      endif
22
23      " back to origin window
24      if a:bang
25          if l:tab != 0 && l:tab != tabpagenr()
26              execute l:tab . 'tabnext'
27          endif
28          if l:win != 0 && l:win != winnr()
29              execute l:win . 'wincmd w'
30          endif
31      endif
32  endfunction "
```

這裏，仍然按短命令思想，定義 ':TC' 用於在內置終端中執行任意命令，就是將其參數用 term_sendkes() 函數轉發給內置終端，並自動添加了回車鍵。':TC!' 加歎號修飾的話，會回到原來的普通窗口。用 ':TCD' 跳到內置終端窗口，並自動將內置終端的當前目錄切到原來編輯文件所在目錄（就是自動執行 cd 命令啦）。因爲 TCD 的用意就是切到內置窗口，並開始在指定目錄下與終端進行交互工作，那肯定是不必跳回原來的，所以傳給實現函數的第一個參數寫定爲 0 。

查找並切到終端窗口的函數，這裏不再列出，主要是通過 &buftype 選項值是否爲 terminal 來判斷。有興趣的可以到這個地址查看詳細代碼。如果不習慣短命令，或擔心命名名衝突，儘可自行改自己覺得滿意的足夠長的命名名，或者再定義個快捷鍵映射。

第九章　VimL 混合編程

9.1　用外部語言寫過濾器

混合編程場景介紹

本章來討論 VimL 與其他語言混合編程的話題。這“混合”編程可能不是很准確的定義，也許涉及不同層面的場景應用。在上一章介紹的異步編程也算是其中一種吧。不過如果所調用的外部程序是別人已經寫好的（或者是系統提供的經典工具），那用戶就只能適應其提供的接口或輸出，在 vim 端幾乎沒什麼可干預的。但如果利用通道連接的另一端的程序，也要自己開發，那就可以從設計開始就考慮如何更好地與 VimL 協作，並且顯然另一端可以使用任何主流語言。這就不再多說了，本章主要着眼於其他場景的（同步）混合編程。

與衆所周知的另一件編輯神器相比，vim 是比較純粹的編輯器，它本身提供的功能（雖然編輯方面非常豐富）比較集中，也就比較依賴或吻合 Unix 哲學：一個工具把自己的事做好，並且便於與其他工具配合。所以，當 vim 想處理更復雜的事務時，它天然地傾向於與其他工具“混用”。比如，從最基本打開文件編輯，vim 也接受從其他工具的管道輸入：

```
$ ls -l | vim -
```

這個命令表示將當前目錄下的文件列表送入 vim 中編輯，譬如打算在每行前面添加 mv 命令，想仔細規劃下如何批量重命名。

就像許多 Unix 工具一樣，啓用 vim 時若用 - 取代文件名參數，就表示從標准輸入讀入內容，所以它很容易配合管道，作爲接收管道輸入的末端。但由於 vim 常規運用是作爲可視化交互式全屏編輯，它不再產生標准輸出，因而也不便繼續產生管道輸出至下一工序。然而，vim 也有批量模式，不會打開交互界面，實際上也是可以強行配合，達到類似 sed 的流編輯效果——但這就似乎有點旁門左道了，不是 vim 的常規用法。

當然，管道的配合，只是工具的組合與混用，離“可編程”的概念還比較遠。

在支持異步的版本之前，system() 函數只能在 VimL 這端進行邏輯與流程控制，而對所調用的外部命令不可控，這可算“半混合”編程。其實還有另一個叫“過濾器”的功能用法，它是允許與鼓勵對所調用的外部腳本進行編程，但在 vim 這端的用法卻是固定的，因而也可算是另一種“半混合”編程。

除了自己寫通道服務算“全混合”編程外，vim 在這之前還提供了多種腳本語言的內

置接口，那也算是（同步的）“全混合”編程了。本節先介紹相對簡單的過濾器，下一節再介紹語言接口。

過濾器的概念與使用

即使你對過濾器並不熟，但也應該用過 = 重縮進命令，那就是個特殊的過濾器。

過濾器的意思是將當前編輯 buffer 中指定範圍的文本，當作標准輸入調用某個外部程序，並用其標准輸出替換原範圍的文本，以此達到修改、編輯的目的。因此，重縮進與格式化的本質也就是過濾器。

過濾器的標准使用方式是在命令行一對地址範圍之後接 ‘!’ 與外部命令，如：

```
:n1,n2 ! 外部程序
:1,$ ! 外部程序
:'<, '> ! 外部程序
:. ! 外部程序
```

注意必須在 ‘!’ 前有地址參數，否則 “! 外部程序” 就是純粹切到外部 shell 運行那個外部程序了。而過濾器並不會打斷用戶切到外部 shell ，只要不是處理巨量文本，替換輸入輸出應該都比較快，雖然是同步，一般沒延遲問題。

如果只有一個地址參數，表示只處理一行，. 表示當前行。如果是兩個地址參數，則表示起始行到終止行的範圍，1,$ 表示從第一行至最後一行，即全部文本。可以在命令行手動輸入兩個數字行號，也在可視模式下選擇一定範圍後按: 自動添加’<,’> 表示所選擇的行範圍。

使用過濾器還有個快捷鍵方式，不必先按 ‘:’ 進入命令行，直接在普通模式按 ‘!’ 再接一個移動命令（文本對象），也會自動幫你選定這個文本對象，並自動進入命令行模式並填充好地址參數，用戶只要繼續在 ‘!’ 之後輸入想調用的外部程序。

誠然，過濾器可以直接調用別人寫好、已經完善的外部程序。然而，由於以標准輸出替換標准輸入的模型如此簡單，而每個人的編輯任務又可能多種多樣各具個性，在一時找不到合用的外部工具時，用戶完全可以用他所熟悉的任一種腳本語言快速寫個過濾器。

比如再舉那個簡單的例子吧，給文本行編號？最簡單的需求，其實可以直接用 cat -n 命令完成：

```
:'<,'>!cat -n
```

注意，從 vim 命令行調用外部過濾器時，可以附加命令行參數傳給過濾器程序，選擇文本是標准輸入，這兩者互無關係。如果要對全文編號，一定別忘了加地址參數: 1,$!; vim 有不少可能作用於範圍的命令，在缺省時默認表示全文，但過濾器若省了地址參數就解釋爲普通 ‘!’ 外部調用命令了。

但是，cat -n 的編號似乎不美觀，右對齊，空白太多。如果你想編號左對齊，數字後面最好還能加個符號，如 1. 或 1) 等，再或者想爲指定行編號，比如跳過註釋行……等等，

不一而足的需求。如果你熟悉某種腳本語言，最好是自己操起腳本語言來寫適合的過濾器。例如，下面這個 perl 腳本，實現爲文本行編號：

```
1 " file: catn.pl
2 my $sep = shift || "";
3 my $num = shift || 0;
4 $sep .= ($num > 0) ? (" " x $num) : "\t";
5 while (<>) { print "$.$sep$_"; }
```

該過濾器腳本接受兩個參數，第一參數指定緊接數字編號的後綴符號，第二參數指定之後隔幾個空白，如果缺省，就隔一個製表符。在 while 循環中，<> 符號用於從標准輸入讀取數據，$. 表示行號，$_ 表示當前行文本，這樣語義就明確了，行號與分隔字符串與原文本拼接起來作爲標准輸出。(用其他語言寫這個腳本也不復雜，只是語法不一樣，總是可以手動累加行號的)

如果將該腳本保存在當前目錄中，可以在 vim 命令行嘗試一下：

```
:'<,'>!perl catn.pl
:'<,'>!perl catn.pl .
:'<,'>!./catn.pl . 2
```

如果給腳本添加了可執行權限，可直接將腳本作爲過濾器程序，否則就將腳本文件當作 perl 解釋器的第一參數。如果腳本不在當前目錄，請替換爲腳本全路徑，或者若將可執行腳本放在某個 $PATH 路徑中，也可以直接使用。然後要注意命令參數，會先後經過 vim 命令行與 shell 命令行兩層處理，對特殊字符最好加引號或轉義，避免出錯。例如：

```
:'<,'>!./catn.pl ')' 2
:'<,'>!./catn.pl '*' 2
:'<,'>!./catn.pl \% 2
:'<,'>!./catn.pl '\#' 2
```

如果) 不加引號，會出現 shell 語法錯誤；如果 * 不加引號，在 shell 中會展開爲當前文件的所有文件名，這可能不是想要的；當然這兩個字符也可以用 \轉義，安全地從 shell 命令行傳入 catn.pl 過濾器腳本。

Vim 命令行中的% 符號會被展開爲當前文件名，即使用引號，也是將文件名字符串括在引號中傳給 shell （如果文件名中有空格，有無引號影響 shell 將其作爲幾個參數），如果要將百分號傳給 shell，就得用 \% 反槓轉義。# 在 vim 命令行中會被展開爲 “上一個編輯過的” 文件名，僅用 \# 可以將 # 傳給 shell ，但在 shell 中這符號是註釋，那又會有問題，所以必須用 ‘\#’ 兩層保護，才能將 # 符號傳入過濾器腳本中，輸出類似 1#，2# 的編號效果。

記不住這許多特殊符號規則怎麼辦，很簡單呀，多試試就好，或者用保守的 ‘\#’ 就差不多了。而且在 vim 試錯了過濾器（參數問題，或腳本本身 bug）不要緊，如果意外修改

了文本，按撤銷命令 u 就好。

當然，有時特意利用 vim 特殊符號的替換意義也可能是有用的，例如你又想文件名放在行號之前了，類似 file:1 的效果。那麼就可以在 vim 命令行中傳入 '%' 參數，如果確認當前文件名中沒空格，也可以不用引號。當然了，這個過濾器腳本本身的邏輯功能也要作相應修改了。

所以你看，只要你經常腦洞大開，需求總是在不斷變化。然而只要掌握一門腳本語言，哪怕只會寫簡單的教科書式的標准輸入輸出的小程序，運用過濾器思維，就能極大地擴展 vim 的編輯效率與趣味性。

9.2 外部語言接口編程

語言接口介紹

Vim 支持其他諸多語言接口。這意味着，你不僅可以寫 VimL 腳本，也可以使用被支持的語言腳本。這就相當於在 vim 中內嵌了另一種語言的解釋器。當然你不能完全像其他語言的解釋器來使用 vim ，畢竟還是遵守 vim 制定的一些規範，那就是 vim 爲該語言提供的接口。

在 Vim 幫助首頁，專門有一段 Interfaces 的目錄，列出了 Vim 所支持的語言接口，大都以 if_lang.txt 命名，其中 lang 後綴指某個具體的（腳本）語言。筆者較熟悉的腳本語言有 lua、python、perl ，而其他如 ruby、tcl 較少了解。因而在本章打算簡要介紹下 if_lua if_python 與 if_perl 這幾個語言接口。（因 python 有兩個版本，故在幫助文檔中其實用 if_pyth.txt 命名，避免 python 狹義地指 python2，不過本文仍習慣使用 python 統稱）

一些功能複雜的插件，爲了規避 VimL 語言的不足，都傾向於按語言接口採用其他語言來完成一部分或主要功能。比如，unite 就採用了 if_lua 接口，後來的升級版 denite 則採用 if_python 接口，另外推薦一個插件 LeaderF 也是用 if_python 寫的。這都是不錯的實際項目源碼，想深入學習的可以參考。

不過採用 if_perl 接口的現代插件較少，筆者鮮有看到。但是筆者偏愛 perl ，所以在本章剩余篇幅將重點以 if_perl 爲主，也算略微彌補一點空白。而且，Vim 爲各語言提供的接口大同小異，思路是一致的。介紹一種語言接口，也期望讀者能舉一反三。真要用好某種語言接口，除了要仔細學習 vim 相關的 if_lang.txt 文檔，還需要對目標語言掌握良好，才能方便地在兩種環境中來回🄵弋。

自定義編譯 vim 支持語言接口

默認安裝的 vim 一般不支持語言接口，需要自己重新從源碼編譯安裝。這也其實很簡單，只要修改一些編譯配置即可。首先從 vim 官網 或其 github 鏡像 下載源代碼包，解壓後進入 src/ 子目錄，vi Makefile 查找並取消如下幾行註釋：

```
CONF_OPT_LUA = --enable-luainterp
CONF_OPT_PERL = --enable-perlinterp
```

```
CONF_OPT_PYTHON = --enable-pythoninterp
```

原來這幾行是被 # 註釋的，表示相關語言接口是被禁用的，你所需做的只是刪去 # 符號啓用功能。當然每個語言接口在 Makefile 都提供了好幾個不同的（被註釋）選項備用，各有不同的含義，典型的如動態鏈接或靜態鏈接。上面示例是打開靜態鏈接編譯選項，含 =dynamic 的表示動態鏈接編譯選項。你只需打開（取消註釋）其中一條選項，一般建議用靜態鏈接編譯。動態鏈接只是減少最後編譯出的 vim 程序的大小，或許也略微減少 vim 運行時所需的內存。在硬盤與內存都便宜的情況下，這都不算問題，用靜態鏈接可減少依賴，避免版本不兼容的麻煩。

不過 python 語言接口分 python2 與 python3 兩個選項，它們既像一個語言又像兩個語言。打開 python3 接口的編譯選項是 –enable-python3interp 。注意，你不能同時打開 python2 與 python3 的靜態編譯選項，如果想同時支持，只能都用動態鏈接編譯選項。除非你有絕對理由想同時使用 python2 與 python3 ，還是建議你只使用其中之一。而且 python2 都是歷史原因，以後的趨勢都應該都是轉向 python3 。

在自定義安裝 vim 時，還有個選項推薦打開，就是安裝到個人家目錄下，不安裝到系統默認的路徑下，也就不影響系統其他用戶使用的 vim 。只要指定 prefix 即可，一般也就是打開（取消註釋）如下這行：

```
prefix = $(HOME)
```

然後，就可以按 Unix/Linux 源碼編譯安裝程序的標准三部曲執行如下命令了：

```
$ make configure
$ make
$ make install
```

如果你運氣足夠好，應該直接 make 成功的。如果 make 失敗，最可能的原因是系統沒有安裝相應的語言開發包，請用系統包管理工具（yum 或 apt-get）安裝語言開發包，如 perl-dev ，注意有些系統爲語言開發包名的命名後綴不同，也可能是 perl-devel 。安裝好了所需語言開發包（及可能的其他依賴），再重新 confire make 應該就能成功了。

在編譯成功之後，make install 安裝之前，最好檢查一下新編譯的 vim 是否滿足你所需的特性。執行如下命令：

```
$ ./vim --version
```

在 vim 命令之前添加 ‘./’ 表示使用當前目錄（src/ 編譯時目錄）的 vim 程序，否則可能會查找到系統原來的 vim 程序。如果打印的版本信息，包含 +perl （或 +lua +python），就表示成功編進了相應的語言接口。當然，你也可以直接不帶參數地啓動 ‘./vim’ 體驗一下，並可在 vim 的命令行查看如下命令的輸出：

```
: version
: echo has('perl')
```

```
: echo has('python')
: echo has('python3')
```

‘:version’ 命令與 shell 命令參數 –version 的輸出基本類似。has() 函數用於檢測當前 vim 是否支持某項特性，如果支持返回真值（1），否則假值（0）。has() 函數也經常用於 VimL 脚本尤其是插件開發中，爲了兼容性判斷，根據是否支持某項特性執行不同的代碼。

確認無誤後，就可以 make install 安裝。所謂安裝也不外是將剛纔編譯好的 vim 程序及其他運行時文件與手冊頁等文件，複製到相應的目錄中。安裝的根目錄取決於之前 $prefix 選項，如果按之前指導選擇了 $(HOME) ，那 vim 就會安裝到 ~/bin/vim 中。一般建議將個人家目錄下的 ~/bin 添加到環境變量 $PATH 之前，這樣在 shell 啓動命令時，首先查找 /bin 目錄下的程序。

當然了，在你決定手動編譯 vim 之前，最好在目前默認使用的 vim 中用 ‘:version’ 與 has() 檢測下它是否已經支持相應的特性了，如果已經支持，那就可跳過這裏介紹的手動編譯流程了。

語言接口的基本命令

測試某個語言接口是否真的能正常工作，也可直接以相應語言名作爲 vim 的命令，執行一條目標語言的簡單語句，例如：

```
: perl print $^V
: perl print 'Hello world!'
: lua print('Hello world!')
: python print 'Hello world!'
: python3 print 'Hello world!'
```

其中第一條語句是打印 if_perl 接口使用的 perl 版本，其後就是使用不同語句打印喜聞樂見的 Hello world! 了。

語言名如 ‘:perl’ 也就是相應語言接口的最基本接口命令了，可見它們保持着高度的一致性，vim 調用相應的語言解釋器執行其參數所代表的代碼段，所不同的只是各語言的語法文法了。下面，如無特殊情況，爲行文精簡，就基本只以 if_perl 爲例說明了。

基本命令 ‘:perl’ 只適合在命令行執行簡短的一行 perl 語句（當然，對於 perl 語言，單行語句也可以很強大）。如果要執行一大塊 perl 語句，適合在脚本中用 here 文檔語法，即 VimL 也像許多語言一樣支持 « EOF 標記：

```
1 perl << EOF
2 print $^V; # 打印版本號
3 print "$_\n" for @INC; # 打印所有模塊搜索路徑
4 print "$_ = $ENV{$_}" for sort keys %ENV; # 打印所有環境變量
5 EOF
```

EOF 只是約定俗成的標記，其實可以是任意字符串標記，甚至可以省略默認就是單個點. 號。Vim 會從下一行開始讀入，直到匹配某行只包含 EOF 標記，將這塊內容（長串字符串）送給 ':perl' 命令作爲參數。換用其他標記的理由，一般是內容本身包含 EOF 避免誤解。

不過良好的實踐，不推薦將 perl « EOF 裸寫在某個 *.vim 腳本文件中，而應該封裝在一個 VimL 函數中，最好再用 if has 判斷保護，如：

```
function! PerlFunc()
    if has('perl')
        perl << EOF
        print $^V;
        print "$_\n" for @INC;
        print "$_ = $ENV{$_}" for sort keys %ENV;
EOF
    endif
endfunction
```

注意：EOF 不能縮進，只能頂格寫，即整行只能有 EOF 才表示 here 文檔結束。這樣封裝之後，更能提高代碼的健壯性與兼容性。然後就可按普通 VimL 函數一樣調用了 ':call PerlFunc()'。

當然，每次都寫 if has 判斷可能有點繁瑣，那麼可以將這個判斷保護提升到更大的範圍內，如：

```
if has('perl')

function! PerlFunc1()
    perl code;
endfunction

function! PerlFunc2()
    perl code;
endfunction

endif
```

或者將所有利用到語言接口的代碼收集到一個腳本，然後在最開始判斷：

```
if !has('perl')
    finish
endif
```

在 if_lua 或 if_python 接口中，還提供執行整個獨立的 *.lua 或 *.py 脚本文件的命令，如下：

```
:luafile script.lua
:pyfile script.py
```

但是比較奇怪，if_perl 並沒有類似的 ‘:perlfile’ 命令，要實現類似功能，可以用 ‘:perl require ”script.pl”’ 命令，並且要注意 perl 的模塊搜索路徑問題。而在 ‘:luafile’ 或 ‘:pyfile’ 命令中，查尋命令行中提供的脚本文件，還是 vim 的工作，取決於 vim 的搜索路徑。

另外一個很有用的命令是 ‘:perldo’，它會遍歷指定當前 buffer 範圍的每一行（默認是 1,$ ），將 perl 的默認變量 $_ 設爲遍歷到的那行文本（不包括回車換行符），如果 ‘:perldo’ 命令參數的代碼段修改了 $_ ，它就會替換 “當前” 行文本。例如：

```
:perldo s/regexp/replace/g
:%s/regexp/replace/g
```

上面兩行語句其實是一樣的意義，都是執行全文正則替換，只不過第一行 ‘:perldo’ 採用 perl 風格的正則語法，它實際執行的是 perl 語句；第二行:%s 就是執行 VimL 自己的正則替換。如果你想體會 perl 正則與 VimL 正則有什麼異同，或對 perl 正則比較熟悉，覺得某些情況下用 perl 正則更舒服，就可以用 ‘:perldo s’ 代替%s 試試。

當然，‘:perldo’ 所能做的事情遠不只 s 替換，s 在 perl 語言中只是一個操作符。perl 語言的單行語句非常強大，尤其是支持後置 if/for/while 的條件判斷或循環，這就取決於用戶的 perl 語言造詣了。

不過 ‘:perldo’ 命令，與上一節介紹的過濾器機制略有不同，嘗試用它實現給文本行編號的功能，最初的想法可能是：

```
:perldo $_ = "$. $_"
```

但這不能達到要求，‘$.’ 在 ‘:perldo’ 遍歷的每一行中都輸出 0 ，這說明 perl 並沒有把文本行當成標准輸入（或其他輸入文件）處理，並沒有給 $. 變量自動賦值。改成如下語句能達到編號需求：

```
:perldo $_ = ++$i . " $_"
```

看起來有點像 perl 的黑魔法，其實不過是藉助了一個變量 $i ，未定義變量當作數字用時被初始化 0 ，然後也支持像 C 語言的前置 ++i 語法，然後又將該數字通過點號 ‘.’ 與一個字符串連接，代表行號的數字自動轉化爲字符串。這樣創建使用的 $i 將是 perl 的全局變量，在執行完這條語句後，可以再用如下語句：

```
:perl print $i
```

查看 $i 的值，可見它仍保留着最後累加到的行號值。如果再次執行上面的 ‘:perldo’ 語句對文本行編號，那起始編號就不對了。需要手動 ‘:perl $i = 0’ 重置編號。但這也正意味着，如果要求編號從任意值開始，上述 ‘:perldo’ 語句就很容易適應。

在 lua 或 python 語言接口中，也有類似 ‘:perldo’ 的命令。但是它們沒有類似 $_ 默認變量的機制，‘:luado’ 與 ‘:pydo’ 實際是在循環中爲每行隱含調用一個函數，傳入 line 與 linenr 參數代表 “當前” 行文本與行號，然後在參數的代碼段中可以利用這兩個參數進行操作，並可用 return 返回一個字符串，取代 “當前” 行。在寫法上沒 perl 那麼簡潔，而且在單行語句中不像函數的地方使用 return 也多少有點違和與🄵戲感。

目標語言訪問 VIM

顯然，如果使用一種語言接口，只是換一門語言自嗨諸如打印 Hello world 這種是沒有前途的。決定使用一種語言接口時，總是期望能利用那種語言更強大的能力，如更快的運算速率或更豐富的標准庫第三方庫功能，完成一系列數據與業務邏輯處理後，最終還是要通過某種形式反饋到 vim ，對 vim 有所影響纔是。

爲此，if_lua 與 if_python 都提供了專門的 vim 模塊，在目標語言中將 vim 視爲一個邏輯對象，可從那語言代碼中直接訪問、控制 vim ，如設置 vim 內 buffer 的文本，執行 vim 的 Ex 命令等。if_perl 也提供類似的模塊，名叫 VIM，使用語法與常規點號調用方法不同而已，perl 使用 ‘::’ 與 ‘->’ 符號。

以 if_perl 爲爲例，其 VIM 模塊提供瞭如下實用接口：

- VIM::DoCommand({cmd}) 從 perl 代碼中執行 vim 的 Ex 命令；
- VIM::SetOption({arg}) 設置 vim 的選項，相當於執行 ‘:set’ 命令；
- VIM::Msg({msg}, {group}?) 顯示消息，相當於 ‘:echo’，但可以指定高亮顏色；
- VIM::Eval({expr}) 在 perl 代碼中計算一個 vim 的表達式；
- VIM::Buffers([{bn}...]) 返回 vim 的 buffer 列表或個數；
- VIM::Windows([{wn}...]) 返回 vim 的窗口表表或個數。

其中，前三個接口方法只是執行 vim 的命令，perl 代碼中不再關注其返回值。後三個方法是計算與 vim 相關的表達式，需要獲得並利用其返回值。而 perl 語言的表達式是有上下文語境的概念的。VIM::Eval() 方法在標量環境中獲得一個 vim 表達式的值，並轉化爲 perl 的一個標量值。所謂 vim 表達式，比如 @x 表示 vim 寄存器 x 的內容，&x 表示 vim 的 x 的選項值。當然簡單的 1+2 也是 vim 的表達式，但這種平凡的表達式直接在 perl 代碼中求值也是一樣的意義，沒必要使用 VIM::Eval() 了。Vim 中的環境變量 $X 也與 perl 中 $ENV{X} 等值。perl 的標量值具體地講就是數字或字符串。但如果該方法在列表語境中求值，則結果也是一個列表，特別地是二元列表：

```
($success, $value) = VIM::Eval(...);
@result = VIM::Eval(...);
if($result[0]) { make_use_of $result[1] };
```

返回結果的第一個值表示 Eval 求值是否成功，畢竟參數給定的 vim 表達式有可能非法，如果成功，第二值纔是實際可靠的求值結果。如果確信求值有意義，可直接用標量變量接收 VIM::Eval() 的返回值，那就是求值結果，可簡化寫法，省略成功與否的判斷。

VIM::Buffers() 與 VIM::Windows() 的上下文語境就更易理解了，它符合 perl 的上下文習慣：本來是數組的變量，在標量上下文表示數組的大小。所以不帶參數的 VIM::Buffers() 返回所有 buffer 的列表，或在標量語境下返回 buffer 數量。如果提供參數（可以一個或多個），就根據參數篩選 buffer 列表。如果想獲取某個特定的 buffer，也得通過在列表結果中取索引，例如：

```
$mybuf = (VIM::Buffers('file.name'))[0]
```

你得保證 file.name 至少匹配一個 buffer，否則返回空列表，再對空列表取索引 [0] 是未定義的值。而且一般建議參數給精確，能且只能匹配一個 buffer，否則如果匹配多個，按 vim 的 bufname() 函數的行爲，在歧義時也返回空。如果給的參數是表示 buffer 編號的數字，一般能保證唯一，只要是有效的 buffer 編號。給這個方法傳多個參數時，就返回相應參數個數的 buffer 列表，例如：

```
@buf = VIM::Buffers(1, 3, 4, 'file.name', 'file2.name')
```

就將取得一系列指定的 buffer 對象，存入於 @buf 數組中。

一旦獲得 buffer 對象，就可以用對象的方法，操作它所代表的相應的 vim buffer：

- Buffer->Name() 獲得 buffer 的文件名；
- Buffer->Number() 獲得 buffer 編號；
- Buffer->Count() 獲得 buffer 的文本行數；
- Buffer->Get({lnum}, {lnum}?, ...) 獲取 buffer 內的一行或多行文本；
- Buffer->Delete({lnum}, {lnum}?) 刪除一行或一個範圍內的所有行；
- Buffer->Append({lnum}, {line}, {line}?, ...) 添加一行多多行文本；
- Buffer->Set({lnum}, {line}, {line}?, ...) 替換一行或多行文本；

Window 對象也有自己的方法，請查閱相應文檔，這裏就不再羅列了。此外，還提供兩個全局變量用於操作當前 buffer 與當前窗口：

- $main::curbuf 表示當前 buffer ；
- $main::curwin 表示當前窗口。

由於‘:perl’命令執行的 perl 代碼，就默認在 main 的命名空間（包）內，所以一般情況下可簡寫爲 $curbuf 與 $curwin 。

9.3 * Perl 語言接口開發

本節將專門講一講 if_perl 接口的開發指導與實踐經驗，雖然只講 perl ，但其基本思路對於其他語言接口也可互爲參照。

VimL 調用 perl 接口的基本流程

典型地，假如要使用（perl）語言接口實現某個較爲複雜的功能或插件，其調用流程大概可歸納如下：

1. 定義快捷鍵映射，nnoremap ，這不一定必要，可能直接使用命令也方便；
2. 快捷鍵調用自定義命令，command；
3. vim 自定義命令調用 vim 自定義函數；
4. 在 vim 函數中使用:perl 命令調用 perl 函數；
5. 在 perl 函數中實現業務運算，可能有更長的調用鏈或引入其他模塊；
6. 在 perl 函數使用 VIM 模塊將運算結果或其他效果反饋回 vim 。

在以上流程中，前三步是是純 VimL 編程（細究起來，前兩步准備動作還只是使用 vim），第 5 步是純 perl 編程，而第 4 步與第 6 步就是 VimL 與 perl 的接口過渡。接口的使用只能按標准規定，打通一種可能，而要直接實現有意義的功能，重點還是迴歸到第 5 與第 3 步兩門語言的掌握程度上。

整個流程是同步的，當 perl 代碼執行完畢後，堆棧上溯，一直回到第 1 步的命令完成，纔算一條 vim 的 Ex 全部完成，然後 vim 繼續響應等待用戶的按鍵。

但凡編程，要有作用域的意識，在這第 4 步中，首先是在 VimL 的函數的局部作用域中，首次進入的 perl 代碼，是在 perl 的 main 命名空間。如果在 perl 的後續調用鏈中，進入了其他命名空間，再想引用本次 vim 命令（第 2 步）或之前 vim 命令中在 perl main 命名空間定義的變量，就得顯式加前綴 'main::' 或簡寫 '::' 也可。在 perl 代碼中，使用 VIM 模塊，只能直接影響 vim 的全局變量，它無法獲知調用 ':perl' 命令所處的函數作用域或腳本作用域。如果有這個需求，請約定使用的全局變量，並在 ':perl' 代碼同步返回時，及時從被影響的全局變量更新局部變量保存下來。

另一個基本意識是有關程序的輸入輸出。從 ':perl' 開始執行的代碼，它的標准輸出被重定向到 vim 的消息區。所以如果打印簡單字符，':perl print' 與 ':echo' 效果差不多。在這裏執行的 perl 不應試圖從標准輸入讀取數據，如果需要輸入，可以打開文件的方式（如臨時文件，或確定的目標文件），或者利用 VIM 模塊直接讀取 buffer 內容。

Perl 代碼與 VimL 代碼解耦

雖然語言接口允許你將兩種語言混用寫在一起，但當真正想實現一些較複雜功能時，將兩種語言的代碼分別保存在獨立的 *.vim 或 *.pl 是更好的代碼維護與項目管理方式。而且也儘量將使用了 VIM 模塊的 perl 腳本與未使用 VIM 模塊的代碼分開。

因爲 VIM 模塊只能是從 vim 執行的 perl 代碼纔可用。將那些未使用 VIM 模塊的純數據運算邏輯的 perl 代碼獨立開來，方便獨立測試，也便於將其複用在非 vim 環境下的常規 perl 腳本開發中。使用了 VIM 模塊的 perl 代碼，只方便在 vim 環境下測試。如果一定要在外部獨立測試調試，只能自己提供一個簡易模擬版的 VIM.pm ，將在腳本用到的 'VIM::' 方法都實現出來（比如就打印調試信息之類）。

如下代碼段可以判斷 perl 是否運行在 vim 環境（是否通過 ':perl' 調用的）：

```
1 package main;
2 our $InsideVim = 0;
3 {
```

```
4       eval { VIM::Eval(1); };
5       $InsideVim = 1 unless $@;
6   }
```

perl 的 eval 語句塊，有類似的 try ... catch 的功能，就是嘗試執行 VIM 模塊的隨便一個有效的方法，最簡單就是 VIM::Eval(1) 了。如果不是從 vim 環境執行，eval 會出錯，出錯信息保存在 $@ 變量中。如果確實在 vim 環境中，eval 正常執行，$@ 爲空，unless 是條件取反，變量 $InsideVim 被置爲 1 標記之。

然後就可以根據 $InsideVim 的值來做分支判斷了。如果代碼只設計在 vim 環境中使用，當 $InsideVim 爲假值時可直接 return 或 exit 。如果特意還是想在非 vim 環境下通過測試，那就可以在 $InsideVim 爲假時引用自寫的簡易調試版 VIM.pm 。

只爲調試用的模擬 VIM 模塊大致結構可以如下：

```
1   # File: VIM.pm
2   package VIM;
3
4   sub DoCommand{
5       my $cmd = shift;
6       print "Will do Vim Ex Command: $cmd\n";
7   }
8
9   sub Eval{
10      my $expr = shift;
11      print "Will eval Vim expression: $expr\n";
12      return $expr;
13  }
```

也許還應該爲 Eval() 函數添加自適應列表環境與標量環境的返回值，還有 Buffer 與 Window 對象的方法，模擬實現都會更復雜。故沒必要求全，只根據實際情況，待測試的腳本用到哪些方法，首先讓腳本能編譯能運行，再考慮進一步模擬精度的必要性。當然最可靠的還是在 vim 中整合起來測試效果，只是在 vim 只能交互地手動測試，有時略有不便。

順便提一下，使用 if_perl 時，不必顯式聲明 use VIM; 就能在相關代碼中使用 VIM 模塊。但使用 if_python ，還是要顯式聲明 import vim 的。

Perl 與 VimL 數據交換的幾種方式

首先，簡單的 perl 代碼，如果 print 至標准輸出的，在被 vim 調用時是打印到消息區的，因而可以用重定向消息的方法，將 perl 的標准輸出內容捕獲至 vim 變量中。例如，專門寫個 ifperl.vim 存些基本工具函數，如：

```
1   " File: ifperl.vim
```

```
function! s:execute(a:code) abort
    let l:perl = 'perl ' .  a:code
    redir => l:ifstdout
    silent! execute l:perl
    redir END
    return l:ifstdout
endfunction
```

這個函數將封裝執行一段 perl 代碼，將其標准輸出當作一個變量返回（爲簡明起見，省略了錯誤等特殊情況處理）。一般更推薦調用 perl 函數，如此利用 s:execute() 也很容易封裝函數調用：

```
function! s:call(func, ...) abort
    let l:args = join(a:000, ',')
    let l:code = printf('%s(%s);', a:func, l:args)
    return s:execute(l:code)
endfunction
```

實際上，在 vim 命令行向 perl 函數傳參數還得注意引號問題，這裏也從略。然後，模擬‘:pyfile’實現並未內置支持的‘:perlfile’功能，也可簡單封裝成一個函數，如果也想關注執行一個 *.pl 可能的輸出，可以改用上面的 s:execute() 函數：

```
function! s:require(file) abort
    execute printf('perl require("%s");', a:file)
endfunction
function! s:use(pm) abort
    execute printf('perl use "%s";', a:pm)
endfunction
function! s:uselib(path) abort
    execute printf('perl use lib("%s");', a:path)
endfunction
```

注意，在 perl 中，require 與 use 語句有區別，各有用途。但都涉及搜索路徑，在程序中推薦用 use lib 動態添加。可以將用於 vim 調用的 perl 腳本收集在一個目錄（或專門的插件目錄），並用 use lib 添加這個目錄，便於 vim 使用。

其次，如果要用到的 perl 腳本，主要是一些工具函數，要利用其返回值的，而不是打印到標准輸出的。這種情況下，若強行在 perl 處加一層打印函數，在 vim 處重定向消息，那是比較低效也不優雅的。另一個可考慮的替代的辦法是專門設計幾個全局變量槽讓 perl 訪問。例如；

```
" File: ifperl.vim
```

```
let g:useperl#ifperl#scalar = ''
let g:useperl#ifperl#list = []
let g:useperl#ifperl#dict = {}
```

```
# File: ifperl.pl
sub ToVimScalar
{
    my ($val) = @_;
    VIM::DoCommand("let g:useperl#ifperl#scalar = '$val'");
}
sub ToVimList
{
    my ($array_ref) = @_;
    VIM::DoCommand("let g:useperl#ifperl#list = []");
    foreach my $val (@$array_ref) {
        VIM::DoCommand("call
            add(g:useperl#ifperl#list, '$val')");
    }
}
sub ToVimDict
{
    my ($hash_ref) = @_;
    VIM::DoCommand("let g:useperl#ifperl#dict = {}");
    foreach my $key (keys %$hash_ref) {
        my $val = $hash_ref->{$key};
        VIM::DoCommand("let g:useperl#ifperl#dict['$key']
                                = '$val'");
    }
}
```

在 perl 中的三種數據類型，標量、列表、散列，分別可對應 VimL 變量的字符串、列表與字典，並且字符串在可能的情況下都可當作數字使用。當 perl 裏的數據需要發往 VimL 時，臨時藉助事先規定好的這幾個全局變量做緩存，只多調用一層轉接函數，不影響原來 perl 函數的使用方式。

最後，其實要考慮的問題，是否真有必要將 perl 數據發還 VimL 。在協作完成一個功能時，得盤算好哪部分必須在 VimL 處完成，哪部分可集中在 perl 處完成，沒必要的中間結果就別傳回 VimL 處理了。

如果真要從 perl 頻繁傳出大量文本，自己用變量接收也不如用 VIM 內部的 Buffer 方

法有效率。例如，也專門設計一個 buffer，取名 IFPERL.buf ，在 perl 中將需要查看的文本直接附加到這個 buffer 的末尾：

```
" File: ifperl.vim
let g:useperl#ifperl#buffer = 'IFPERL.buf'
```

```
# File: ifperl.pl
sub ToVimBuffer
{
    my $bufname = VIM::Eval('g:useperl#ifperl#buffer');
    my $buf = (VIM::Buffers($bufname))[0];
    $buf->Append($buf->Count(), @_);
}
```

這裏直接將 ToVimBuffer() 的參數全部傳給 Append() ，便支持同時添加多行（字符串列表）或一行（標量字符串）至 vim buffer 中。須提醒的是 Append() 方法的第一參數，不能使用 '$' 表示最後一行，只能是數字，因爲這是在 perl 代碼中，'$' 沒有特殊行號意義，當作普通字符串轉化爲數字時，就是 0 ，結果就會添加到 buffer 最前面而不是最後面。

這種策略也適於記錄被 vim 調用的 perl 代碼執行過程的日誌，直接發到某個 vim buffer 中查看。在開發調試時有奇效，比寫日誌文件更有效，然後由用戶再決定有無必要保存日誌。當然，完整的日誌功能需要更靈活的控制，如在生產中就應該關閉。

小結

使用 if_perl 接口混合編程的一個實用示例可參考這個插件：useperl。本節上述引用的代碼段也多是從該插件簡化而來的。該插件目前主要利用了 if_perl 實現 perl 語言編寫補全，理論上利用 vim 內嵌的 perl 解釋器可達到語義理解級別，只是在具體實現細節上還比較初步，可能不甚完善。

然後說明一個事實，vim 支持多種語言接口，直接原因並非 VimL 本身設計多厲害（vim 的厲害之處更在其他整體綜合上），而是因爲那些腳本語言設計良好，方便嵌入其他程序。例如，perl 與 python 都可提供 C/C++ 擴展，而 vim 就是個 C 語言寫的應用程序；還有 lua 語言最初設計目的就是便於嵌入到其他更大型的程序或服務上。所以 vim 利用這些腳本語言的開發接口，編入它們的解釋器，原非大驚小怪。也許，vim 還正想借這些語言彌補自 VimL 腳本的不足。

那麼，有了這些語言接口，是否就弱化了 VimL 腳本的意義了呢。那也不盡然，有些功能還是適合用 VimL 來實現，尤其是涉及用戶界面接口部分，如快捷鍵 noremap 與自定義命令 command 還有 GUI 版本的菜單 menu 。此外，VimL 兼容與移植性更好，畢竟其他語言接口不是默認編譯選項。使用統一的官方 VimL 語言更有利於用戶的交流與融合。

所以，隨着 vim 的進化與發展，VimL 語言也應該穩步發展，這將成爲 vim 文化與社區不可或缺的一部分。

第十章　Vim 插件管理與開發

10.1　典型插件的目录規範

學 VimL 脚本的終極目標是寫插件按需擴展 vim 的功能。在開始着手寫插件之前，有必要先了解一下典型的、功能較齊全的插件，應該如何組織目錄結構，按 vim 的習慣將不同類別的功能放在相應的子目錄下。

vim 運行時目錄

插件的目錄，可參考 vim 本身安裝的運行時目錄。所謂運行時目錄，顧名思義，就是在 vim 運行時如果要加載 *.vim 脚本，應該到哪裏找文件。

有兩個相關的環境變量，可用如下命令查看：

```
:echo $VIM
:echo $VIMRUNTIME
```

如果從源碼安裝 vim ，且自定義安裝於家目錄的話，它們的值大概如下：

```
1 $VIM =  ~/share/vim
2 $VIMRUNTIME = ~/share/vim/vim81
```

所以 $VIM 指的是 vim 安裝目錄，而且不同版本的 vim 都將安裝在該目錄下，$VIMRUNTIME 就是具體當前運行的 vim 版本的安裝目錄。不過此安裝目錄不包括 vim 程序本身 (那是被安裝到 ~/bin 中的)，主要是 vim 運行時所需的大量 *.vim 脚本，相當於 “官方插件”。該目錄有哪些文件目錄，可用如下命令顯示：

```
:!ls -F $VIMRUNTIME
```

就是 shell 的 ls 命令，選項 -F 只是在子目錄後面添加 / ，使得容易區分子目錄與文件。也許直接從 shell 執行 ls 是被 alias 定義的別名，自動加上了一些常用選項，但從 vim 內用! 調用是不讀別名的。

$VIMRUNTIME 既是官方目錄，顯然是不建議用戶在其內修改或增刪的。如果不是自定義安裝在個人家目錄，使用系統默認安裝的 vim 的話，普通用戶也無權修改。

於是 vim 提供了一個選項叫 &runtimepath （常簡稱 &rtp），那是類似系統 shell 的環境變量 $PATH，就是一組目錄，只不過不用冒號分隔，而是用逗號分隔。可用如下命令查看 &rtp：

```
:echo &rtp
:echo split(&rtp, ',')
```

通常，~/.vim/ 目錄會在 &rtp 列表中，而且往往是第一個。另外，官方目錄 $VIMRUNTIME 也在 &rtp 列表較後一個位置。當 vim 在運行時需要加載腳本時，就會依次從 &rtp 列表中每個目錄（及其子目錄）中查找，有時查找第一個就會停止。所以 $VIMRUNTIME 目錄並不特殊，只是 &rtp 中一個優先級並不高的目錄。對用戶來說，~/.vim/ 目錄才更特殊些，常被稱爲 vim 的用戶目錄。

一般建議用戶將個人的 vimrc 及其他 vim 腳本放在 ~/.vim/ 目錄中。可以用這個命令：

```
:echo $MYVIMRC
```

查看當前你運行的 vim 啓動時讀取 vimrc。如果顯示是 ~/.vimrc ，則建議將其移至 ~/.vim/vimrc 或軟鏈接指向它。vim 會嘗試讀取 vimrc 的幾個位置及順序，也可用如下命令查看：

```
:version
```

然後提一下，如果是 windows 操作系統，沒有 ~/.vim/ 目錄。但它肯定有 $VIM 安裝目錄，然後用戶目錄就是 $VIM/vimfiles 。

當了解了用戶目錄 ~/.vim/ ，就可以參照官方目錄 $VIMRUNTIME 來組織管理自己的 vim 個性化配置及擴展腳本（插件）。

全局插件目錄 plugin/

最簡單的插件就是將 *.vim 腳本存到（某個）&rtp 的 plugin/ 子目錄下。當 vim 啓動時，就會讀取（每個）&rtp 的 plugin/ 子目錄下的 *.vim 腳本並加載。因爲它們總是被加載，故有時稱爲全局性插件。

一般 vimer 初學階段，傾向於完善與豐富自己的配置 vimrc 。當 vimrc 文件越來越大感覺不便維護時，可將部分功能拆成獨立腳本放在 plugin/ 目錄下，畢竟這個目錄下的腳本也是能初始加載的，與合在 vimrc 中沒有太大區別。可以想象一下，常規 vimrc 配置大約有如下內容：

- 使用 set 設置的選項
- 使用 map 系統列定義的快捷鍵
- 使用 command 定義的命令
- 自動事件命令組 augroup
- 自定義函數

- 爲 gVim 定義的菜單
- 其他

如果爲以上某部分內容進行了重度自定義，譬如快捷鍵，對鍵盤上每個按鍵都仔細自己規劃了一遍，甚至需要一些簡單函數以便支持快捷鍵功能；那麼就可嘗試將這部分抽出來，另存爲名如 ~/.vim/plugin/myremap.vim 的腳本。極端點，可以將 vimrc 中每部分功能都拆出來扔到 plugin/ 目錄。而 vimrc 只需留下這兩行：

```
1 set nocompatible
2 filetype plugin indent on
```

這就是網上曾流傳的所謂 “最簡配置”。第一行設置爲不兼容 vi 模式，意即開啓 vim 的擴展功能；第二行是打開文件類型檢測。另外我還建議在 vimrc 中定義一個環境變量 $VIMHOME 保存用戶目錄：

```
1 let $VIMHOME = $HOME . '/.vim'
2 if has('win32') || has ('win64')
3     let $VIMHOME = $VIM . '/vimfiles'
4 endif
```

這樣，在之後的 vimrc 或其他腳本的代碼中，引用 $VIMHOME 就更有通用性，尤其是在需要手動加載（:source）腳本時。

不管是從大 vimrc 拆出腳本，還是從頭開始寫某個功能腳本放在 plugin/ 目錄，都要注意全局插件的一些特性。

其一是某個 plugin/ 目錄下的所有 *.vim 腳本加載順序不能保證。因此每個腳本要相應獨立完成某個或某類功能，避免引用其他兄弟腳本定義的全局變量。如有這需求，類似 $VIMHOME 環境變量，還是在 vimrc 中定義吧，保證最開始被執行到。

其次是 plugin/ 的所有腳本還包含其子目錄，即更深層次下的 &rtp/plugin/**/*.vim 腳本也會被自動加載。利用這個特性，可以對該目錄進一步組織管理，將相關門類功能的腳本再放入更恰當的子目錄名。但也要避免這個特性濫用，太深層次目錄搜索比較耗時，可能會影響 vim 的啓動速度。故一般不建議在 plugin/ 下再建子目錄，最多再建一層。

如果 plugin/ 中腳本太多，影響 vim 啓動速度，應該將其移出 plugin/ 目錄。可能的直覺錯誤是在 plugin/ 下建個 backup/ 子目錄，把某些不想用但想備用的腳本扔進去，這不管用，藏不住的。可以把 *.vim 腳本後綴改爲 *.vim.bak ，這就不會被 vim 啓動加載了。更好的建議是建一個與 plugin/ 平級的 plugin.bak/ 子目錄，因爲文件後綴名對 vim 編輯是重要的。

順便說一下，在 vim 啓動時，也有命令行參數可以指示 vim 在啓動時跳過加載 plugin/ 的腳本。但一般日常使用時不必考慮這種差別。

類型插件目錄 ftplugin/

與全局插件相對應的，是局部，具體講，是與某種文件類型相關的插件，只在打開對應類型的文件時才生效。

文件類型是 vim 的一個概念，每個編輯的文件，都有個獨立的選項值 &filetype，這就是該文件的類型。直觀地看，文件名後綴代表着其類型。但本質上這不是同一個概念。vim 只是主要根據文件名後綴來判斷一個文件類型，有時還根據文件的部分內容（如前幾行）來判斷文件類型，用戶還可以用 set filetype= 來手動設置一個類型。一種文件類型也可以關聯好幾個後綴名，比如 cpp、hpp 都是 C++ 文件，文件類型都是 cpp，同樣情況還有 htm 與 html 後綴名的文件，都認爲是 html 文件類型。

文件類型插件要生效，還得在 vimrc 中添加 filetype plugin on 這行配置，這一般也是推薦必須配置。然後在打開文件併成功檢測到屬於某種文件類型時，vim 就會加載 &rtp/ftplugin/&ft.vim 腳本。

例如，每當打開 *.cpp 或 *.hpp 文件時，vim 都認爲它屬於 cpp 文件類型，它就會加載 ~/.vim/ftplugin/cpp.vim 腳本，以及其他 &rtp 目錄下的 ftplugin/cpp.vim 。實際上，vim 搜尋文件類型插件腳本時規則很寬松，還會嘗試搜索 cpp_*.vim 腳本，甚至子目錄 cpp/*.vim 下的腳本。這目的是允許在同一個 ftplugin/ 目錄中爲一種文件類型提供多個插件腳本，它們都會被加載運行。

相比於 plugin/ 目錄中的插件腳本只會在 vim 啓動時執行一次，ftplugin/ 則可能在 vim 運行時重複執行多次。每打開相應類型的文件（准確地說是 &filetype 選項值被設置時觸發）就會再次搜索並執行所有 &rtp/ftplugin 中所有匹配類型的腳本。

因此爲了避免無意義重複工作，在文件類型插件腳本中，只推薦寫那些確實每個文件（buffer）都需要獨立設置的工作，如：

- setlocal 設置局部選項值
- remap 系列命令加上 <buffer> 參數，只爲當前文件定義快捷鍵
- command 自定義命令也加上 -buffer 參數
- let 命令只修改 'b:' 作用域的變量

此外，還可以在相應的腳本中，通過 VimL 語法來控制腳本的實際執行。比如，參考官方目錄的 cpp 類型插件，使用 ':e $VIMRUNTIME/ftplugin/cpp.vim' 打開，內容如：

```
1 " Only do this when not done yet for this buffer
2 if exists("b:did_ftplugin")
3   finish
4 endif
5
6 " in c.vim
7 " let b:did_ftplugin = 1
8
9 " Behaves just like C
```

```
10 runtime! ftplugin/c.vim ftplugin/c_*.vim ftplugin/c/*.vim
```

開始幾行通過判斷 b:did_ftplugin 變量的存在性來決定是否繼續加載當前這個腳本，一般在加載當前腳本時會將該值設爲 1 ，這是 vim 官方推薦的文件類型插件的標准頭寫法。注意如果每個類型插件都是這樣寫，那是排他的意義，那就是加載了其中第一個類型插件的腳本，就不會再加載其他（有這個保護頭的）腳本。雖然 vim 的機制會繼續搜索其他匹配的類型插件腳本，但 VimL 語句層面上控制了不會重複加載，而這種控制是用戶可選的方案。

最後一行表示 cpp 類型“繼承”加載所有 c 類型的插件腳本，這是符合 C++ 語言與 C 語言特定業務關係的。這樣就可以將 C/C++ 相關的都只寫在 c.vim 類型插件中，避免重複代碼。事實上，那個 b:did_ftplugin 變量就只在 c.vim 中定義，不能在 cpp.vim 前面先定義，否則執行到 c.vim 是會被跳過。

有時在類型插件腳本中，比如定義局部快捷鍵時，不可避免要到調用特定函數以便封裝具體實現。這種函數顯然也只應該隨文件類型插件加載，沒用到過該類型就沒必要加載，但是與局部快捷鍵需要爲每個新打開文件定義的情況不同，函數定義最好只定義一次，不必爲每個新文件重複定義。

如果是自己寫在 ~/.vim/ftplugin/&ft.vim 中，腳本大致結構可以如下：

```
1  if exists("b:dotvim_ftplugin")
2    finish
3  endif
4  let b:dotvim_ftplugin = 1
5
6  " 設置局部選項、快捷鍵等
7
8  if exists("s:dotvim_ftplugin")
9    finish
10 endif
11 let s:dotvim_ftplugin = 1
12
13 " 剩余只需加載一次的支持函數、代碼
```

注意開頭使用 b:dotvim_ftplugin 變量控制，不同於官方習慣的變量 b:did_ftplugin，主要是不想有排他性。也就是說自己只想在 ~/.vim 用戶目錄下額外加些設置，執行完後還想加載官方的（或安裝在其他目錄的第三方的）同類型插件。

同樣地，也可以在用戶目錄中讓一種文件類型繼承加載另一種文件類型。但是‘:runtime’命令太泛了，會搜索所有 &rtp 目錄。我們自己明確知道另一個目標文件類型是哪個腳本，就直接用‘:source’會更有效率，例如在 ~/.vim/ftplugin/cpp.vim 中：

```
source $VIMHOME/ftplugin/c.vim
```

當然了，按個人實際情況，很可能都不會寫純 C 代碼，那就直接維護 cpp.vim 腳本好了，不必額外有個 c.vim 腳本。另外，也有可能不同的文件類型都有部分共同設置代碼，那也可以提取出來放在獨立的 ftplugin/language.vim 腳本中，然後在各個具體的文件類型插件腳本中都調用這個腳本：

```
source $VIMHOME/ftplugin/language.vim
```

這裏假設沒有哪種文件類型名恰好叫 language，不過若防意外，也可以故意取個比較特殊的名字，如 ftplugin/__common__.vim。

文件類型其他相關目錄

與文件類型相關的目錄，不止 ftplugin/ 這一個。ftplugin/ 一般是通用目的的 VimL 代碼，還有其他幾個目錄，是 vim 爲了實現其他具體功能時所需讀取的腳本，雖然它們也是 *.vim 後綴名的腳本，理論上也可以寫任意 VimL 代碼，但實踐習慣上只爲完成特定功能。

因本書的主旨是講 VimL 的，所以對這些目錄或文件只簡單羅列介紹於下：

- syntax/ 定義文件類型的語法高亮規則，基於正則匹配的；
- compiler/ 定義相應語言的編譯命令及錯誤格式
- indent/ 設定縮進規則
- filetype.vim 檢測文件類型的規則，自動事件 filetypedetect
- indent.vim 設置自動縮進的事件
- ftplugin.vim 文件類型插件加載機制

如果閱讀這些官方腳本的源碼，就會發現 ftplugin.vim 等就是利用自動事件實現的。顯然也可以自己在 vimrc 中用 autocmd 實現根據文件後綴名加載特定的相關腳本。但是由於這個需求如此常見，官方已經幫我們做好了，並且支持了大量你見過的與未見過的編程語言。

另外，類似全局插件功能的，除了 plugin/ 外，也還有其他幾個約定目錄。如 colors/ 就是定義配色主題的。這裏就不一一介紹了。

自動加載目錄 autoload/

autoload/ 是放自動加載腳本的目錄，在第 5.5 節介紹自動加載函數時就已提及。不過由於它在現代 vim 中非常重要，故這裏再單獨列出。自動加載機制是順應 vim 發展而提出的，也是 VimL 腳本語言的一大進步，因爲 autoload/ 就相當於 perl/python 等腳本語言存放模塊的搜索路徑。自動事件（autocmd）是 vim 內置機制，用戶無法過多干涉，autoload/ 自動加載函數是自動事件的一個重要擴充，允許用戶在 VimL 語言層面對函數與腳本的自動加載作靈活的控制。

自動加載函數是名字中含有 # 的函數，如 part1#part2#final()，其函數名代表着（某個 &rtp 目錄下的）autoload/ 目錄下的相對路徑，如 autoload/part1/part2.vim 。基於這

種對應關係，定義自動加載函數的腳本不必在 vim 啓動時事先加載，可以在 vim 運行時直接調用，首次調用時就會從 &rtp 中找到相應的腳本自動加載。當然這是按 &rtp 順序找到的第一個自動加載腳本就採用，所以 ~/.vim/autoload 往往有最高的優先級。但最好避免這種潛在的命名衝突與隱藏。

關於自動加載函數的用法，請回顧複習第 5.5 節，這裏不重複了。不過全局變量名也可以採用 # 的標記，如 g:part1#part2#varname，只在取值時會觸發自動加載。

一般在開發較大型插件時，應該將主要實現函數都放在 autoload/ 目錄下，並且建議將插件名再建一層子目錄，這樣該插件使用的函數名都有相同的前綴，或可稱爲命名空間。而在 plugin/ 與 ftplugin/ 目錄中只寫簡單的用戶界面如快捷鍵、命令定義。如此在一定程度上就相當是 vim 接口與 VimL 實現的分離，有利於大型插件的項目管理。

善後目錄 after/

after/ 是個很有趣的目錄，每個 &rtp 目錄下的 after/ 子目錄又是一個 &rtp 目錄，被自動添加到原來常規的 &rtp 列表之末。該 after/ 目錄的結構可以與其父目錄或其他 &rtp 目錄一樣。如果你瞭解數學上“分形”這個概念，可作此類比理解，就是“部分與整體擁有相似的結構”。

如果使用‘:echo &rtp’命令，很可能在回顯消息的末尾看到如下兩個目錄：

```
$VIMRUNTIME/after
$VIMHOME/after
```

因爲 after/ 是自動添加到 &rtp 列表末尾，而 vim 在搜索運行時腳本時按順序搜索 &rtp ，所以 after/ 目錄可以保證儘可能後地被搜索。這機制有什麼用途呢?

運行時腳本有兩類明顯不同的搜索方式。一種是搜索第一個匹配的腳本就停止，如 autoload/ 目錄下的腳本，如此排在 &rtp 前列的具體更高的優先級。另一種是始終搜索所有 &rtp 目錄，如 plugin/ 與 ftplugin/ ，如此排到 &rtp 末尾的腳本具有更高的優先級。

如果用戶安裝了許多插件，每個插件被安置在獨立的 &rtp 目錄中（詳見下一節的插件管理），那麼不同 &rtp 目錄下的同名腳本，就有可能衝突。因此在本插件目錄下另建 after/plugin/ 或 after/ftplugin/ 目錄可以大概率保證本插件提供的功能不被覆蓋。

但是，一般的插件，除非有特別理由，不建議添加 after/ 子目錄。強行武斷地排他，提升自己的優先級。最好尊重用戶的意願，保留用戶目錄 $VIMHOME/after/ 讓用戶自己決定如何解決衝突，覆蓋其他插件的影響。

同時，也不要故意爲難 vim，在 after/ 目錄下繼續遞歸地建立 after/ 目錄。

文檔目錄 doc/

最後要介紹的文檔目錄。vim 提供了詳盡的在線使用手冊，或叫幫助文檔。在使用過程中如有任何疑難雜症，都推薦使用‘:help’嘗試。如果英文水平有限的，可以下載一份

中文翻譯文檔。但最好還是習慣英文原文文檔，畢竟命令與函數名是沒辦法翻譯成中文的，熟悉 vim 官方文檔使用的術語，有助於更好使用 vim。

官方文檔放在 $VIMRUNTIME/doc 目錄下，就是 txt 純文本文檔。不過有特殊的約定格式，尤其是表示超鏈接目標與跳轉到超鏈接的表示法，其他語法顏色對於 vim 已是司空見慣。

用戶可以並且建議爲自己開發的插件編寫文檔，放在自己的 $rtp/doc 目錄下，然後用‘:helptags’生成索引（需要指定 doc/ 目錄作爲參數），以便支持跳轉，這樣就納入了 vim 的幫助文檔系統。用不帶參數的‘:help’打開幫助系統首頁，在末尾部分有一節名爲 LOCAL ADDITIONS 的，就列出了本地幫助文檔，也就是除 $VIMRUNTIME 以外的其他 &rtp 目錄下的 doc/*.txt 文檔。

最後提一句，善用幫助文檔是學習與使用 vim 的不二法門。看過的任何書籍或技術博客文章，都大概率看過就忘記的，包括你正在看的這一本，它們的價值在於領進門，幫忙建立個概念，在實際遇到問題時還知道個搜索關鍵字，或者是‘:help’的主題參數。至於詳細使用細節，都以 vim 幫助文檔爲准。

10.2 插件管理器插件介紹

插件管理的必要性

上一節介紹了 vim 用戶目錄（~/.vim，並推薦設爲 $VIMHOME 環境變量）。這在自己寫簡單腳本是很方便的，按規範將不同性質與功能的腳本放在不同子目錄。但這有個潛在的問題，源於你不能總是自己造輪子，且不論是否有能力造複雜的輪子。

這世界上多年以來有許多狂熱的 vim 愛好者，開發了許多優秀的插件，應該善加選擇然後下載安裝使用。但是如果都安裝到 ~/.vim 目錄，那來源於不同插件的腳本就混在一起了，既可能造成腳本同名文件衝突，也不利於後續維護（升級或卸載）。

後來，vim 提供了一種 vimball 的安裝方式。就是將一個插件的所有腳本打包成一個文件，其實也是符合 VimL 語法的腳本，直接用 vim 的‘:source’命令，就會把“包”內的文件解壓出來，放到 ~/.vim 目錄下，並跟蹤記錄解壓了哪些文件，哪個文件來源於哪個安裝包，然後將來要升級替換或下載刪除時便有🄵可尋。但這仍不可避免來源於不同插件的腳本同名衝突，且將個人用戶目錄搞得混雜不堪，對有潔癖的 vim 用戶尤其是程序員是不能忍受的。

再後來，隨着 github 的流行，版本控制與代碼倉庫的概念深入人心，vim 的插件使用與管理思想也發生了革命性的變化。其實原理也很簡單，關鍵還是 &rtp ，那不是一個目錄，而是一組目錄，除了官方 $VIMRUNTIME 與用戶目錄 $VIMHOME 外，還可以將任意一個目錄加入 &rtp 列表。因此，可以將每個來源於第三方的插件放在另外的一個獨立目錄，在該目錄內也按 $VIMRUNTIME 目錄規範按腳本功能分成 plugin/、ftplugin/ 等子目錄，再將其添加到 &rtp 中。如此，在 vim 運行時也就能在需要時從這個目錄讀取第三方插件內的腳本，就和安裝（拷貝）到 $VIMHOME 下的效果一樣。只是現在每個插件

都有着自己的獨立目錄，甚至可直接對應 github 倉庫，升級維護變得極其方便了。

在所謂的現代 vim 時代，“插件”這詞一般就特指這種有獨立目錄並按 vim 規範組織子目錄的標准插件，插件內的各子目錄的文件一起協作完成某個具體的擴展功能。而之前那個用戶目錄 $VIMHONE ，建議只保留給用戶自己保存 vimrc 及其他想寫的腳本，不再被安裝任何第三方插件。在這之前，$VIMHOME 目錄尤其是 plugin/ 子目錄下的每個腳本，也許都被稱爲一個插件，爲了區分，或可稱之爲“廣義的插件”。而從現在開始，單說插件時，只指“狹義”的標准插件（目錄）。

當安裝插件變得容易時，安裝的第三方插件就會越來越多，這時又誕生了另一個需求。那就是如何再管理所有這些第三方插件？本章的剩下內容就來介紹一些便利的插件管理工具及其管理思路。這些插件管理工具本身也是個第三方插件。

pathogen

首先介紹一款插件 pathogen。看其名字 pathogen，大約是“路徑生成器”的意思，其主要工作就是管理 vim 的 &rtp 目錄列表。

按 pathogen 的思路，是在 $VIMHOME 約定一個 bundle/ 子目錄，然後將所有想用的第三方插件下載安裝到該目錄。對於託管在 github 上的插件，可以直接用 git clone 命令，例如就安裝 pathogen 插件本身：

```
$ mkdir -p ~/.vim/bundle
$ cd ~/.vim/bundle
$ git clone https://github.com/tpope/vim-pathogen
```

這樣，插件安裝部分就完成了，可以如法炮製安裝更多插件。然後要 vim 識別這些插件，讓它們真正生效，還要在 vimrc 中加入如下兩句：

```
1 set rtp+=$VIMHOME/bundle/vim-pathogen
2 execute pathogen#infect()
```

其中第一句是將 pathogen 本身的插件目錄先加到 &rtp 列表，如此才能調用第二句的 pathogen#infect() 函數。很顯然，這是個自動加載函數，它會找到 autoload/pathogen.vim 腳本，加載後就能正式調用該函數了。該函數的功能就是掃描 bundle/ 下的每個子目錄，將它們都加入 &rtp 列表，就如同第一句顯式加入那樣；只不過 pathogen 批量掃描，幫你做完剩余的事了。

事實上，pathogen 插件只有 autoload/pathogen.vim 這一個腳本是關鍵起作用的，如果將該文件安裝（下載或拷貝）到 $VIMHOME 中，那就沒必要第一句顯式將 pathogen 加入 &rtp ，因爲它已經能在 &rtp 中找到了。如果在 Linux 系統，若爲安全或潔癖原因，不想複製污染用戶目錄，則可用軟鏈接代替：

```
$ cd ~/.vim/autoload
$ ln -s ../bundle/vim-pathogen/autoload/pathogen.vim pathogen.vim
```

所以 pathogen 插件本身不必安裝在 bundle/ 目錄中，bundle/ 只是它用來管理其他後續安裝的第三方插件。如果不想混在個人用戶目錄中，pathogen 可以安裝在任意合適的地方，只要在 vimrc 將其加入 &rtp 或如上例做個軟鏈接。

vundle 與 vim-plug

pathogen 在管理 &rtp 方面簡單、易用，且高效、少依賴。只有一個缺點，那就是還得手動下載每一個插件。如果連這步也想偷懶，那還有另一類插件管理工具可用，如下幾個都支持自動下載插件：

- Vundle
- vim-plug
- dein

其中，Vundle 出現較早，自動安裝的插件默認也放在 ~/.vim/bundle 目錄，只不過需要在 vimrc 中用 ‘:Plugin’ 命令指定要使用的插件。現在基本推薦使用 vim-plug 代替 Vundle ，用法類似，只不過使用更短的 ‘:Plug’ 命令，而且支持並行下載，所以首次安裝插件的速度大大增快。dein.vim 管理插件則不提供命令，要求直接使用 dein#add() 函數，並且插件安裝目錄默認不放在 ~/.vim/bundle 了。

這裏僅以 vim-plug 這個插件爲例介紹其用法。它只有單腳本文件，官方建議安裝到 ~/.vim/autoload/plug.vim 中。但正如 pathogen.vim 一樣，你可以放到其他位置，只是首先在手動維護這個插件管理的插件本身的 &rtp 路徑。然後在 vimrc 進行類似如下的配置：

```
1 call plug#begin('~/.vim/bundle')
2 Plug 'author1/plugin-name1'
3 Plug 'author2/plugin-name2'
4 call plug#end()
```

顯然，plug#bigin() 與 plug#end() 是來自 autoload/plug.vim 腳本的函數，用於該插件管理工具進行內部初始化等維護工作，其中 plugin#begin() 函數可指定插件安裝目錄。然後在這兩個函數調用之間，使用 ‘:Plug’ 命令配置每個需要使用的插件。參數格式 author1/plugin-name1 表示來源於 plugin-name1 的插件。‘:Plug’ 還支持可選的其他參數，比如用於配置複雜插件下載後可能需要進行的編譯命令，這就不展開了。

在 vim 啓動時，vim-plug 會分析 vimrc 配置的插件列表，如果插件尚未安裝，則會自動安裝，並打開一個友好的窗口顯示下載進度及其他管理命令。如果在 vim 運行時編輯了 vimrc ，修改了插件列表，並不建議重新 ‘:source $MYVIMRC’，而是可以手動使用 ‘:PlugInstall’ 命令安裝插件。一般只有在修改了插件列表配置後首次啓動 vim 時纔會觸發自動下載的動作，當然下載速度取決於個人的網絡環境，不過由於它的並行下載，橫向對比其他插件管理工具的下載速度要快。在安裝完插件之後啓動 vim 顯然就幾乎不會影響啓動過程了。當然需要更新已安裝插件時，可用 ‘:PlugUpdate’ 命令。

vim-plug 這類插件管理工具的最大優點是功能豐富，不僅維護插件的 &rtp 路徑，還集成了插件的下載安裝。當插件來源於 github 時，使得插件安裝過程對於用戶極其方便。

相比於 pathogen ，它不僅是替用戶偷懶免去手動下載過程，更簡化了用戶移植個人 vim 配置。比如想將自己的 vim 配置環境從一臺機器挪到另一臺機器，那隻要備份 vimrc 而已（或 ~/.vim 目錄），而插件列表內置在 vimrc 中，可不必另外備份，在新機器上首次啓動 vim 時自動安裝。

Vim8 內置的 packadd

從 Vim8 版本開始，也提供了自己的一套新的插件管理方案，取代曾經🄵花一現的 vimball 安裝方式。核心命令是‘:packadd’，而方案的名詞概念叫 package ，可用‘:help’命令查看詳細幫助文檔。

Vim8 內置的插件管理在思想上更接近 pathogen ，就連 pathogen 的作者也在該項目主頁上推薦新用戶使用 Vim8 的內置管理方案了。因爲這更符合 Vim 一貫以來的 Unix 思維，集中做好一件事。從 Vim 的角度，插件管理就只要維護好每個插件的 &rtp 路徑就盡職了。至於插件是怎麼來的，怎麼下載安裝的，是用 git clone 還是 svn co ，或是手動下載再解壓，再或是用戶自己在本地寫的……Vim 全不管，它只要求你把插件放到指定的目錄，就如一開始規定得把（廣義的插件）腳本放在 plugin/ 目錄一樣。

事實上，之前的 vimball 就是 Vim 曾經試圖介入用戶安裝插件過程的一種嘗試。但是 vimball 沒有成功推廣，僅管那方案有可取之處。所以 Vim8 汲取經驗教訓，package 方案不再糾結用戶安裝插件的事了，用戶愛怎麼折騰就怎麼折騰。

現在，就來具體地介紹 package 方案如何做插件何管理的工作。如果將 pathogen 管理的插件遷移到 Vim8 的 package ，可用如下命令：

```
$ mkdir -p ~/.vim/pack/bundle/start

# 移動 bundle 目{\CJKfamily{fangsong}🄵}
$ mv ~/.vim/bundle/* ~/.vim/pack/bundle/start/

# 爲兼容原目{\CJKfamily{fangsong}🄵}名，建個軟鏈接
$ ls -s  ~/.vim/pack/bundle/start ~/.vim/bundle
```

這裏，~/.vim/pack 叫做一個 &packpath 。那也是 Vim8 新增的一個選項，意義與 &runtimepath 或 &path 類似，是一個以逗號分隔的目錄列表。

我們將 packpath 譯爲包路徑，其下的每個子目錄叫做一個“包”（package），每個包下面可以有多個插件（plugin）。如果包內的插件設計爲需要在 vim 啓動時就加載，就將這類插件放在包下面的 start/ 子目錄中，如果不想在 vim 啓動時就加載，就將插件放在包下面的 opt/ 子目錄中。此後，在 vim 運行時，可以用‘:packadd’命令將放在包下面的 opt/ 類插件按需求再加載起來（直接在命令行手動輸入‘:packadd’命令，或在 VimL 腳本中調用該命令）。

也就是說，在 vim 啓動時，會搜索 ~/.vim/pack/*/start/* ，將找到的每個目錄都認爲是一個標准插件目錄，加入 &rtp 列表並加載執行每個 plugin/*.vim 腳本。當使用‘:packadd

plug-name' 時，就搜索 ~/.vim/pack/*/opt/plug-name/ 目錄，如果找到，則將其加入 &rtp 並加載其下 plugin/*.vim 腳本。然後，&packpath 也不一定只有 ~/.vim/pack/ 一個目錄，用戶可另外設置或增加包路徑。

Vim8 將插件分爲 start/ 與 opt/ 兩類，這是容易理解的。因爲 vim 要優化啓動速度，允許用戶決定哪些插件得在啓動加載，哪些可稍後動態加載。那爲何又要在這之上再加個包的概念呢。那估計是前瞻性考慮了，預計將來 vim 用戶普遍會安裝很多插件，於是可以進一步將某些相關插件放在一個包內，以便管理。

對用戶來說，如何使用包呢? 如果從 github 下載的插件，很容易對應，就將作者名作爲包名，於是該作者的所有插件都歸於一個包。不過對個人用戶來說，極可能只會用到某個作者的一個插件，並不會對他的所有插件都感興趣；況且作者本身也可能只會公佈少數一到兩個 vim 插件。這樣，每個包下面的插件數量太少，又似乎失去了包的初衷意義，而且包下面深層次目錄只有一個子目錄，利用率低，不太 “美觀”。

於是另有一個思路是根據插件功能來劃分包名。想探尋某個功能，找到了幾個來自不同作者開發的插件，各有優劣與適用場景，或者就是要多個插件配合使用，那就可以將其歸於一個包。例如，上面介紹的插件管理插件，出於研究目的，可以都將其下載了，統一放在名爲 plug-manager 的包內：

```
$ mkdir -p ~/.vim/pack/plug-manager
$ cd ~/.vim/pack/plug-manager
$ mkdir opt
$ cd opt
$ git clone https://github.com/tpope/vim-pathogen
$ git clone https://github.com/VundleVim/Vundle.vim
$ git clone https://github.com/junegunn/vim-plug
$ git clone https://github.com/Shougo/dein.vim
```

顯然，這些插件應該歸於 opt/ 類，不能在啓動時加載。事實上，對個人用戶而言，始終建議將下載的第三方插件安裝在 opt/ 子目錄下。否則，啓動時自動加載的 start/ 插件可能太分散，也不利於維護。自己寫的插件，放在 start/ 相對來說更爲穩妥可信，但爲了統一，也建議就放 opt/ 。確定需要啓動加載的，就在 vimrc 中顯式地用 ‘:packadd’ 列出來。或者可以將這樣的一份插件列表單獨放在 ~/.vim/plugin/loadpack.vim 腳本中：

```
1 packadd plugin-name1
2 packadd plugin-name2
3 ...
```

可見，這樣的一份插件列表，就很接近 vim-plug 管理工具要求的 ‘:Plug’ 配置列表了，只是沒有下載功能而已。也許將來，會有配合內置 package 的插件管理工具，用來增補自動下載的功能，供用戶多個選擇。

只是 package 方案出爐略晚，很多用戶已經習慣了 vim-plug 這類的插件管理方式，短

時間內不會轉到內置 package 來。但是之前 pathogen 的用戶，強烈建議改用 package 包管理機制。

10.3 插件開發流程指引

標准插件開發

寫插件有兩個目的，其一是自用，擴展或增強某個功能，或簡化使用接口。其二是發佈到網站上給大家分享。Vim 的官網可以上傳插件壓縮包，不過現在更流行 github 託管。如果僅是給自己用，插件腳本可以寫得隨意點，一些函數的行爲也可以只接受自己選定的一種固定實現。

但如果有較強烈的分享意願，則應該寫得正式點，這是一些實踐總結的建議：

• 遵守第 10.1 節介紹的目錄規範。

• 除非是單腳本插件放在 plugin/ 目錄中，較大型的插件如有多個腳本，則將主要函數實現放在 autoload/ 子目錄中，並且以插件名作爲腳本名，或以插件名再建個子目錄，如此插件名就相當於自動加載函數的命名空間。不過單腳本插件由於具有自包含、無依賴特性，在某些情況下也是方便的。

• 給用戶提供配置參數（全局變量）定製某些功能的途徑，變量名要長，包含插件名前綴然後接具有自釋義性的多個單詞，用下劃線 - 或 # 分隔。並提供文檔或註釋說明。

• 如果插件的主要功能是提供了大量快捷鍵映射，最好爲每個鍵映射設計 <plug> 映射，這種映射名應該與配置變量一樣要長，包含插件名前綴，名字要能反映快捷鍵想做的工作。

• 最好在 doc/ 提供詳盡的幫助文檔，要符合 help 文檔格式規範。在文檔中要說明命令、快捷鍵等用法，及配置變量的意義。文檔也應該隨腳本更新。

• 如果發佈在 github 上，要提供一個 readme.md 說明文檔，除了功能簡介，至少包含安裝方法與安裝命令，便於讓用戶直接複製到命令行或 vimrc 配置中。

插件配置變量

支持用戶配置全局變量的代碼一般具有如下形式，在用戶未配置時設定合理的默認值：

```
1 if !exists('g:plugin_name_argument')
2     let g:plugin_name_argument = s:default_argument_value
3 endif
```

如果要設置默認值的可配置全局變量數量衆多，則可以將這三行代碼封裝成一個函數，讓使用處精簡成一行。設置默認值的函數示例如下：

```
1 function! s:optdef(argument, default)
2     if !has_key(g:, a:argument)
3         let g:{a:argument} = a:default
```

```
4     end
5 endfunction
```

還可以將所有默認值存入 's:' 作用域內的一個字典中，鍵名與全局變量名一致。這樣還能進一步方便集中管理默認值及設置默認值。

然後向用戶說明，哪些快捷鍵是必須在加載插件之前（在 vimrc 中）設定值的，哪些快捷鍵是可以在使用中修改即生效的。

很多插件還習慣用一個 g:plugin_name_loaded 變量，來指示禁用加載該插件，在 plugin/ 腳本的開始處寫入如下代碼：

```
1 if exists('g:plugin_name_loaded')
2     finish
3 endif
```

雖然依 vim 的插件加載機制會讀取到這個腳本，但依用戶的變量配置，有可能跳過加載該腳本的剩余代碼，不再對 vim 運行環境造成影響。

插件映射設計

爲了允許用戶自定義快捷鍵，一個簡單的方法是使用 <mapleader> ，讓用戶可按其習慣使用不同的 mapleader 。另一個更復雜但完備的做法是設計 <plug> 映射。當前有許多優秀插件的映射名使用類似 <plug>(PlugNameMapName) 的形式，把映射名放在另一對小括號中，看起來像個函數名。如果要僞裝成函數，還可以就這樣定義：

```
nnoremap <plug>=PlugName#MapName()
  / :call PlugName#MapName(default_argument)<CR>
```

理解 <plug> 映射的關鍵，就是把 <plug> 當作類似 <CR> 、<Esc> 這樣表示的一個特殊字符好了，只是它特殊到根本不可能讓用戶從鍵盤中按出來。這樣讓 <plug> 作爲插件映射的 mapleader 就不可能與普通映射衝突了。爲了也避免與其他插件的映射相F突，還在 <plug> 字符之後加上表示插件名的一長串字符以示區別。

爲了直觀類比，再想象一下 vip 這個鍵序表示什麼意義？就是依次按下 v i p 三個鍵，它會選定當前段落（空行分隔）！假如要開發一個插件，擴展 vip 選段落的功能（主要目的還應是操作段落），例如根據文件或上下文語境，段落有不同的含義，不一定是空行分隔呢。那麼該快捷鍵映射顯然不能直接覆蓋重定義 vip ，否則用戶 v 進行可視選擇模式會存在困難。至少應該定義爲 <mapleader>vip 。對大部分用戶來說，mapleaer 就是反斜槓，於是按 \vip 就觸發該插件智能選段落的功能。

但這還不夠靈活，更專業的做法是用 <plug>vip ，明示它來源於插件映射。但 <plug> 映射不是給用戶最終使用的接口，因爲 <plug> 字符根本按不出來。所以要雙重映射：

```
nnoremap <plug>vip :call PlugName#SelectParagraph()<CR>
nmap <maplead>vip <plug>vip
```

注意第二個只能用 map 命令，不能用 noremap 命令，因爲它要求繼續解析映射。以上兩行的組合效果相當於是：

```
nnoremap <mapleader>vip
  / :call PlugName#SelectParagraph()<CR>
```

那爲何要多此一舉？程序界有句俗話，很多麻煩的事情，多加一層便有奇效。vim 有個函數 hasmapto() 可判斷是否存在映射，在開發的插件若支持用戶自己定義映射，就該像全局變量配置那樣，判斷用戶自己是否自定義過該快捷鍵了，只在用戶未自己定義時，才提供插件的默認映射。例如：

```
1 if !hasmapto('<plug>vip')
2     nmap <maplead>vip <plug>vip
3 endif
```

所以，讓映射（特別是非內置的插件映射）有個純粹的名字會方便很多。若直接以鍵序如 vip 指代一個映射功能，顯得很詭異，程序可讀性也不高。既然 <plug> 映射主要是作爲名字指稱之用，不是讓用戶直接按的，那它的名字就可以更長更具體些，也可以再加些修飾符號（只要不是空格，否則讓 map 命令解析麻煩）例如：

```
nnoremap <plug>=PlugName#SelectParagraph()
  / :call PlugName#SelectParagraph()<CR>
nmap <maplead>vip <plug>=PlugName#SelectParagraph()
```

當然 PlugName 要替換爲實際爲插件取的名字。至於是否要在前後加 = 與 () 則無關緊要，只是風格而已。常見的風格還有將左括號 (緊接 <plug> 之後，括起整個映射名。但 <plug> 字符必須在映射名最前面。

插件的功能實現最終一般會落實到函數中，所以將插件映射名對應實現的函數名也是良好的風格，方便代碼管理。但由於函數可以寫得更通用些，可以接受參數調整具體功能，而快捷鍵映射沒有參數的概念，所以不能強求映射名與函數名一一對應，而應該爲每個常用參數的函數調用分別定義映射。例如，想用 \Vip 實現與 \vip 不同的功能：

```
nnoremap <plug>(PlugName#SelectParagraphBig)
  / :call PlugName#SelectParagraph('V')<CR>
nmap <maplead>vip <plug>(PlugName#SelectParagraphBig)
```

雖然在插件映射名中也可以加括號與參數表示鍵序以求與函數調用外觀一致，但未必更直觀，而且傳入多個參數時要注意不能加空格。例如：

```
nnoremap <plug>=PlugName#SelectParagraph(Big)
  / :call PlugName#SelectParagraph('V')<CR>
nnoremap <plug>=PlugName#SelectParagraph('V')
  / :call PlugName#SelectParagraph('V')<CR>
```

```
nnoremap <plug>=PlugName#SelectParagraph(1,'\$')
  / :call PlugName#SelectParagraph(1, '\$')<CR>
nnoremap <plug>=PlugName#SelectAll()
  / :call PlugName#SelectParagraph(1, '\$')<CR>
```

從對比中可見，當用參數 (1, '$') 調用函數時，不如直接取名爲 SelectAll 更簡潔易懂。

插件命令設計

插件映射 <plug> 的設計頗有些精妙，在早期的插件中推薦用得比較多。後來自定義命令 command 越來越強大，於是在映射之外，再給用戶界面提供一套命令接口也是一個選擇。

如果將前面的 <plug> 映射名，去掉 <plug> 前綴（對用戶使用來說，也相當於改爲 ':' 前綴）及其他符號，命名或許可再略加省略簡化，那就F身轉變爲了合適的自定義命令名。當然相應地 map 要改爲 command 命令，並注意不同的參數用法。

使用命令作爲函數的用戶接口，很容易實現傳入不同的參數。因此更適合於那些不是非常常用的功能，沒必要分別設計 <plug> 映射，畢竟命名也是椿麻煩事。

爲了使命令更易用，務必提供合適的補全方法。命令自帶提示記憶功能，也是它優於映射的一大特點。命令定義比普通映射覆雜些，但理解起來不比 <plug> 映射困難。

提供了命令及相關說明文檔之後，記得友情提醒一下用戶，讓用戶知道可以自行、任意爲他自己常用的命令定義快捷鍵映射，並自行解決可能的快捷鍵衝突。當然最好也提供一份快捷鍵定義示例，讓用戶可以拷入 vimrc 。例如：

```
nnoremap \vip :PNSelectPargraph<CR>
nnoremap \Vip :PNSelectPargraphBig<CR>
```

而這兩個命令的定義，是寫在插件腳本中的，可以像這樣：

```
command PNSelectPargraph call PlugName#SelectParagraph()
command PNSelectPargraphBig call PlugName#SelectParagraph('V')
```

對於這個 vip 的例子，最後再提一句。直接將其定義爲普通模式的快捷鍵不算是好的設計，那應該是操作符後綴（operator-pending）模式映射，那樣就不僅支持 vip ，還同時支持類似 dip 與 cip 等快捷鍵。不過本章只專注講插件總體設計，就不深入具體實現細節了。

自動加載插件重構

大量安裝插件的新問題

由於插件管理工具的進化，安裝插件變得容易了，一些狂熱用戶就很可能傾向於搜尋安裝過量的插件，啓動 vim 加載幾十上百個插件，並且讓運行時目錄 &rtp 迅速膨脹。雖

然沒有明確的數據顯示，vim 加載多少個插件纔算“過多”，纔會顯著影響 vim 啓動速度以及運行時在 &rtp 中搜索腳本的速度，僅從“美學”的角度看，太長的 &rtp 就顯得笨拙，不夠優雅了。

讓我們直觀地對比下其他腳本語言如 perl/python 的模塊搜索路徑，典型地一般就五、六個，不超過十個。然而 vim 若加載 100 個插件，每個插件按標准規範佔據一個獨立的 &rtp 目錄，那運行時搜索路徑就比常規腳本語言多一個數量級了。(雖然從 vim 使用角度看，似乎包路徑 &packpath 對應着常規腳本語言的模塊搜索路徑，但從 vim 運行時觀點看，搜索 VimL 腳本卻是從 &rtp 路徑中的)

而且 vim 插件的規模與質量參差不齊，除了幾個著名的插件，大部分插件其實都是“簡單”插件，也就是隻有少量幾個 *.vim 文件，甚至就是追求所謂的單文件插件。那麼爲了一兩個腳本，建立一整套標准目錄，似乎有點大材小用。

上節介紹的 dein.vim 插件管理工具也意識到了這個問題，所以它提出了一個“合併”插件的概念，以便壓縮 &rtp 。其實合併插件思想也很簡單，有點像迴歸 vimball 的意味。只不過原來的 vimball 的是無差別地將所有插件“合併”到用戶目錄 $VIMHOME ，如此粗暴地入侵用戶私人空間，僅管有監控登記在案，那也是不足取的。

所以，更溫和點方案是專門另建“虛擬插件”目錄，按標准插件的目錄規範組織子目錄，然後將其他第三方“簡單”插件的腳本文件複製到該目錄的對應的子目錄中（尤其是 plugin/ 內的腳本)。這就實現了合併插件，所有被合併的插件共享一個 &rtp 目錄。而那些著名的大型插件，顯然是值得爲其獨立分配一個 &rtp 的。至於如何判定“簡單”插件，那又是另一個層面的管理策略了。然而如何爲被合併的插件保持可更新，那也是另一個略麻煩的實現細節。

不過，類似 dein.vim 的插件管理工具實現的合併插件，有點像亡羊補牢的措施。作爲插件開發者，可以從一開始就考慮這個問題。如何組織插件結構可使得插件可合併，易於與其他插件共享 &rtp ？這裏就提供一個以此目的的重構思路。

基於自動加載機制重構插件

仍以上述 vip 插件爲例。首先我們爲這個插件確定一個名字，不如簡單點就叫 vip 吧，這插件名字也足夠高大上有吸引力。如果按標准插件規範，這整個插件應該位於 $VIMHOME-/bundle/vip 或 $VIMHOME/pack/bundle/opt/vip 。再假設這是從一個簡單插件開始的，目前主要只有 plugin/vip.vim 這個腳本。

首先，我們將 plugin/vip.vim 腳本移動到 autoload/vip/ 目錄下，並改名爲 plugin.vim :

```
$ mkdir -p autoload/vip
$ mv plugin/vip.vim autoload/vip/plugin.vim
```

然後，編輯原腳本但改名後的 autoload/vip/plugin.vim ，在其末尾增加一個 plugin#load() 函數，空函數即可，或返回 1 假裝表示成功:

```
1 function! plugin#load()
```

```
    return 1
endfunction
```

現在有什麼不同呢？假設原來的 vip/ 插件目錄已被加入 &rtp 中。那麼移動改名之前的 plugin/vip.vim 會在 vim 啓動時加載，而移動改名後的 autoload/vip/plugin.vim 並不會啓動加載。但是可以通過調用函數（手動在命令行輸入或在其他控制腳本中寫）':call vip#plugin#load()' 加載。這個函數名意途非常明確，足夠簡明易懂。如此觸發腳本加載後，原來 vip 插件的所有功能也就加載進 vim 了，其中的命令與快捷鍵映射也就能用了。

既然現在 vip 插件的加載可由用戶 VimL 代碼主動控制了。那就可以將 autoload/vip 這個子目錄複製到其他任意 &rtp 中。當然不建議複製到 $VIMHOME 中。可以單獨建個目錄用於“合併插件”，比如 $VIMHOME/packed ：

```
$ cd ~/.vim
$ mkdir packed
$ cp -r bundle/vip/autoload/vip packed/autoload/
```

在 Linux 系統，也可以用軟鏈接代替複製，只要注意以後所指目標不再隨意改名：

```
$ ln -s ~/.vim/bundle/vip/autoload/vip ~/.vim/packed/autoload/vip
```

然後，在 vimrc 或其他可作爲管理控制的腳本中，加入如下配置：

```
:set rtp+=\$VIMHOME/packed
:call vip#plugin#load()
```

如果有其他插件要合併入 packed/ 目錄，依法炮製即可。將要加載的“插件”調用其相應的 #load() 函數，就如那些插件管理工具配製的插件列表。

自己要開發新插件，也可以從開始按這套路來，都不必另建插件目錄，只要在自己的 $VIMHOME/autoload 建個子目錄，寫個 plugin.vim 腳本，腳本內定義一個 #load() 函數。

但是，如果想分享自己的插件，如何兼容之前的“標准”插件呢。或者說，就這個被改裝重構的 vip 插件，如何回到兼容舊版本呢？那也很簡單，plugin/vip.vim 腳本文件被移走了，再建個新的就是，但是隻要如下一行代碼：

```
" File: plugin/vip.vim
call vip#pligin#load()
```

這樣就可以了，用戶（或者利用某個插件管理工具）可以像標准插件一樣安裝。如果介意 &rtp 路徑膨脹（或其插件管理工具能識別），只要將 autoload/vip 目錄複製到用戶自己選定的另一個合適的共享 &rtp 即可。

簡單插件擴展開發

原來本意爲單腳本的插件，如果後來需要擴充功能，以致代碼量上升，感覺塞在一個文件不太方便時，按標准插件的規範建議，也是將函數拆出來放在 autoload/ 目錄中。

而如果像這裏重構的 vip 插件，本來就是將主體腳本放在於 autoload/vip/plugin.vim 中，在該目錄中添加與 plugin.vim 文件同層級的“兄弟”腳本，那顯然就更加自然了。事實上，更合理的做法正是將插件的具體功能實現分別拆出放在不同腳本中。例如將選擇段落的功能放在 select.vim，將插入段落的功能放在 insert.vim，替換段落的功能放在 replace.vim 中。當然，如何對插件功能抽象，是另一個層面的設計問題，與具體的插件及其規模有關。也許這幾個插件適合都放在一個名爲 operate.vim 的腳本，又或許更復雜的功能適合繼續建子目錄。

這裏的關鍵只是想強調，不要將具體的功能實現（函數）放在 plugin.vim 中。plugin.vim 原則上只寫有關用戶界面操作接口的定義。如 command 定義的命令，map 系列定義的快捷鍵映射。而且，<plug> 插件映射的定義也最好不要暴露在 plugin.vim 腳本中，它們應該定義在相關實現腳本中。plugin.vim 腳本只定義用戶映射，即 <plug> 插件可出現在 map 命令的第二參數中，不可出現在第一參數中。

當插件功能豐富起來後，就要向用戶提供一些（全局變量）配置參數了。然後這些變量參數配置在哪裏也是值得考慮的事了。傳統習慣中，是簡單地讓用戶配置在 vimrc 中。但可想而知，當安裝了許多插件後，你的 vimrc 很可能有大量代碼在配置插件了。此後若刪減或更換了插件，vimrc 中隨意添加的插件變量配置也要記得刪除。否則留下無意義代碼，降低 vim 啓動速度，污染全局變量空間，雖然那程度或許不算嚴重，但想想總是不爽不美的事。

參考加載插件的 vip#plugin#load() 函數，我們也可以相應地設計一個加載配置的函數：vip#config#load()。這就意味着還有個 autoload/vip/config.vim 腳本與 plugin.vim 腳本並列。在這個 config.vim 腳本中，只使用簡單的 let 命令定義插件可用的配置變量的默認值，外帶一個可空的 #load() 函數。真正有意思的是，允許並建議、鼓勵用戶在其私人目錄中提供自己的配置腳本，如 $VIMHOME/autoload/vip/config.vim 。由於個人 $VIMHOME 目錄一般在 &rtp 最前面，這個腳本如果存在的話會優先調用，否則就調用（被合併的共享 &rtp 目錄下）插件的默認配置。雖然，這句配置加載的調用函數應該寫在 plugin.vim 的開始處。於是 plugin.vim 腳本的總體框架現在大約如下：

```
1 " File: vip/plugin.vim
2 call vip#config#load()
3
4 " map 映射定義
5 " command 命令定義，調用其他 vip# 函數
6
7 function! vip#plugin#load()
8     return 1
```

```
9 endfunction
```

如果沒有在當前目錄提供默認的 config.vim ，或擔心用戶提供的 config.vim 腳本忘了定義 vip#config#load() 函數，爲避免報錯，可以將‘:call vip#config#load()’這句調用包在 try ... catch 中保護。

讓用戶將插件配置在獨立的 config.vim 中顯然只應該是建議性的，而不應是強制性的。如果用戶在 vim 啓動時始終要加載的插件，相關插件配置被分散到 autoload/ 目錄下各個 config.vim 小文件中，反而會降低 vim 啓動速度，不如將這些插件配置集中放在一個大文件如 vimrc 中。事實上，用戶將各插件的全局變量配置放在哪裏，並無影響，只是開發者要注意到這個現象。

這個 plugin.vim 腳本的體量可以很少，加載速度可以很快。關鍵在於定義命令時，調用其他 # 函數實現功能，就能在首次調用命令時觸發加載插件中其他相關腳本。而快捷鍵映射，也建議定義爲對命令的調用。如果習慣於 <plug> 映射，則將 <plug> 映射本身定義爲對具體 # 函數的調用（需要隨 plugin.vim 加載，不能像 # 函數自動加載）

用戶在配置自己的 config.vim 時，可以推薦先從插件目錄複製默認 config.vim 到個人目錄，在那基礎上調整自己的參數值。如果變量名取得好，並且有一定的註釋，那該配置文件也自帶文檔功能。更進一步，更激進的點是，如果 plugin.vim 腳本也足夠簡明，只定義命令與映射的話，用戶也可以像複製 config.vim 一樣複製到個人目錄 $VIMHOME 對應目錄下，然後直接修改快捷鍵定義(的第一參數)! 這比在配置中約定一個諸如 g:plugin_name_what 的全局變量更直接。

文件類型相關插件

現在再來考慮文件類型相關的插件，這種可能需要多次加載的“局部”插件，比只需要一次加載的“全局”插件會複雜點。

假設我們這個 vip 插件要支持 cpp 文件類型了，它認爲對於 C++ 源文件來說，什麼叫“段落”應該有它自己特殊的處理。原則仍然是將所有運行時腳本放在 autoload/vip 目錄下。與 plugin.vim 腳本相對應的，文件類型相關功能可以建個 ftplugin.vim 。然後在該腳本中設計一些意途明顯的函數，如 vip#ftplugin#onft()，或者若該插件只想支持少數幾種文件類型（大部分情況如此），直接定義 vip#ftplugin#onCPP() 函數。在該函數內的語句只設置局部選項與局部映射等，供每次打開相應文件類型的 buffer 時調用。而局部映射可能需要用到的支持函數，可直接在 ftplugin.vim 腳本中定義，也能保證只加載一次。

然後，如果 vip 還想兼容標准插件目錄，那就再建個 ftplugin/ 子目錄，其中 cpp.vim 文件只需如下一行調用：

```
:call vip#ftplugin#onCPP() " 或 vip#ftplugin#onft('cpp')
```

如果該插件想合併入共享 &rtp 目錄，則指導用戶將這行語句附加到個人目錄的 $VIMHOME-/ftplugin/cpp.vim 中。一般而言，如果用戶常用 cpp 文件類型，關注 cpp 文件編輯，就該在個人目錄建立這個文件，總有些自己想調教的選項或快捷鍵可以寫在這個腳本中進行

定製。然後安裝的其他能增強擴展 cpp 功能的插件，若都像 vip#ftplugin#onCPP() 這個顯式地在此加行配置，那對 cpp 的影響一目瞭然，也很好地體現了個人目錄腳本的主控性，還能方便切換啓用或禁用某個插件對 cpp 文件類型的影響。

於是，在首次打開某個 *.cpp 文件時，會觸發 autoload/vip/ftplugin.vim 腳本的加載。會保證此時 vip/plugin.vim 腳本已加載，最好在 ftplugin.vim 腳本開頭也加入一行加載插件的調用語句。於是該腳本大致結構如下：

```
" File: vip/ftplugin.vim
call vip#plugin#load()

function! vip#ftplugin#onft(filetype, ...)
    if a:filetype ==? 'cpp'
        return vip#ftplugin#onCPP()
    endif
endfunction

function! vip#ftplugin#onCPP()
    " setlocal ...
    " map <buffer> ...
    " command -beffur ...
endfunction

function! vip#ftplugin#load()
    return 1
endfunction
```

但是如果要支持 vim 默認不能識別的文件類型，這樣就不夠了。例如這個 vip 插件還想自創一種新文件類型，不如也叫 vip 吧，認爲如 *.vip 或 *.vip.txt 後綴名的文件算是 vip 類型。因爲不能識別，所以不會自動加載 ftplugin/vip.vim 腳本。文件類型的檢測是基於自動事件機制，因此可以直接在 vip/plugin.vim 腳本中用‘:autocmd’命令添加支持：

```
"File: vip/plugin.vim
augroup VIP_FILETYPE
    autocmd!
    autocmd BufNewFile,BufRead *.vip,*.vip.txt
            setlocal filetype=vip
augroup END
```

定義了這個事件檢測後，再打開 *.vip 文件 vim 就會自動加載 &rtp/ftplugin/vip.vim 腳本，可在其中調用‘:call vip#ftplugin#onVIP()’，就如支持標准類型 cpp 那樣。但是也

可以直接在‘:autocmd’事件定義中直接調用函數，沒必要間接通過 ftplugin/vip.vim 標准文件類型插件腳本來調用。可改爲如下：

```
1 "File: vip/plugin.vim
2 augroup VIP_FILETYPE
3     autocmd!
4     autocmd BufNewFile,BufRead *.vip,*.vip.txt
5             call vip#ftplugin#onVIP()
6 augroup END
```

其實對於標准文件類型如 cpp 也可以通過類似定義事件調用 vip#ftplugin#onCPP()，但是不要在 ftplugin/cpp.vim 對該函數同時調用了，否則重複調用浪費工作。

插件自創文件類型還有一種典型情形是，該插件有功能打開一個類似管理或信息窗口時，想將該特殊 buffer 設爲一種新文件類型，便於定義局部快捷鍵或語法高亮着色等。這種 buffer 還經常是 nofile 類型，不與硬盤文件關聯，也不存盤。這時就不適合用 autocmd 根據文件後綴名來檢測文件類型了。但是，由於這種 buffer 窗口是完全在腳本控制下創建打開的，直接設定 &ft 就行了。例如，我們的 vip 插件還在某個情況下打開一個提示窗口，不妨將其文件類型設爲 tip ，於是在創建這種特殊 buffer 的代碼處，直接多加兩行：

```
1 " 創建 tip buffer 窗口
2 setlocal filetype=tip
3 call vip#ftplugin#onTIP()
```

注意，當把 &filetype 設爲 tip 時，vim 也會自動去所有 &rtp 搜索 ftplugin/tip.vim 腳本。你可以利用或避免這種特性，決定是否要加 setlocal 這行。而 vip 本身這個插件，對 tip 窗口初始化的入口函數，也像其他標准文件類型一樣，集中放在 vip/ftplugin.vim 中定義。

其他標准插件目錄的考量

以上，在將 vip 插件重構的過程中，將傳統標准插件的 plugin/ 與 ftplugin/ 子目錄移到 autoload/ 下以插件名命名的子目錄中，通過將插件名作爲一級命名空間，來實現插件的動態加載，可達到加速 vim 啓動速度，精簡合併共享 &rtp 的目的。這幾乎可以涵蓋 95% 以上功能拓展型的 vim 插件。

當然也有些特殊目的的插件不適合於 autoload 重構，比如定製顏色主題的 colors/，還有語法定義的 syntax/ 。理論上來說，語法也是文件類型相關的插件，也可以類似地移入 autoload/vip/syntax.vim 文件，將爲每種文件類型定義語法的‘:syntax’語句封裝爲函數，並由 ftplugin.vim 的相應函數來調用。但可能會有可用性與兼容性的問題。除非是插件內自創的臨時文件類型如 tip 需要簡單高亮時，可以考慮直接寫在 vip#ftplugin#onTIP() 函數中（或由這個函數調用他處定義的語法支持）。

此外，還有 doc/ 幫助文檔。這對用戶使用參考很重要，但對 vim 運行時不重要，因此

不在重構範圍內。就仍按標准獨立插件提供文檔吧，如果需要合併插件，也直接複製 doc/ 文檔到共享 &rtp 目錄，也是簡單的。

最後，想說明的是，這裏所討論的“重構”，主要是指插件開發思想上的重構。對於現存寫好的插件，沒太大必要如何折騰，除非有相關插件管理工具能較智能地判斷簡單插件而自動合併與維護。更關鍵的是對於今後開發新插件，無論大小，簡單或複雜的插件，都可以按這思路與規範，儘量將主體腳本封裝在一個 autoload/ 子目錄中，以求最大化地追求支持動態或自動加載，也爲合併插件共享 &rtp 打開方便之門。

筆者有個自寫插件的集合，就在以此思路寫了一些符合自身的實用插件。並且提供一個 ‘:PI’ 短命令，用於簡化手動調用 ‘:call xxx#plugin#load()’ 的加載插件操作。

小結

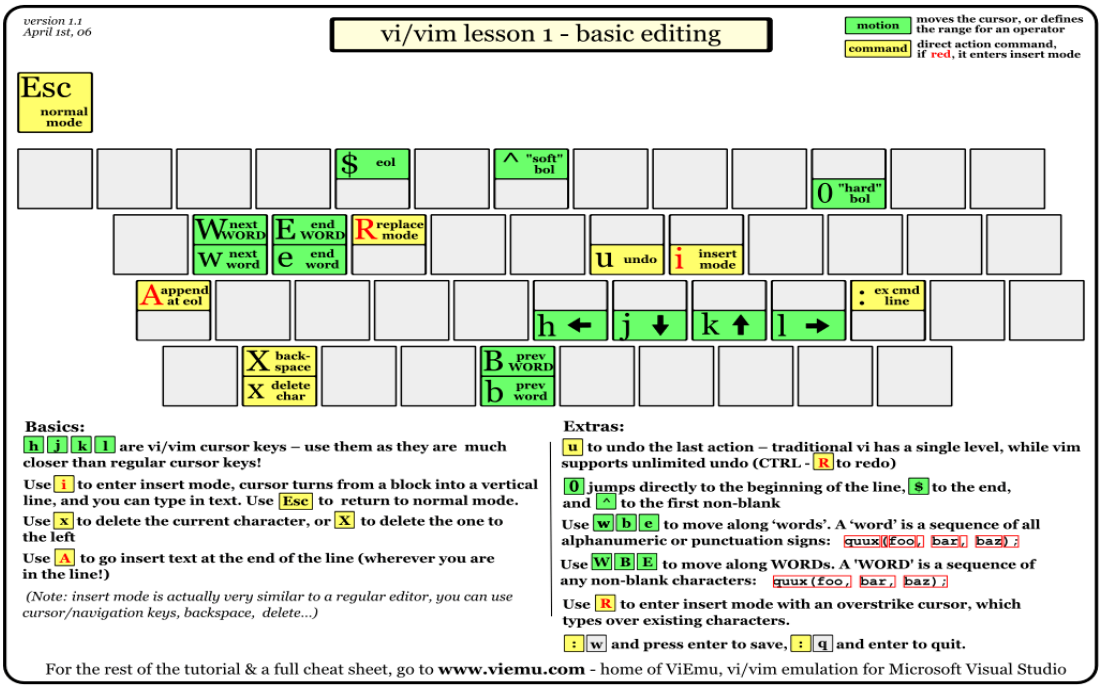
本節介紹了兩種插件開發的範式，一是繼承傳統，一是展望未來。傳統的標准插件，主要依靠 vim 內置固定的幾種機制，在不同的時機去指定的目錄搜尋加載腳本。而後一種自動加載插件，更准確地說是可控的動態加載插件，則主要利用了 VimL 的一種通用的自動加載函數機制，能讓開發者向用戶提供更靈活的插件加載方式與配置方式。

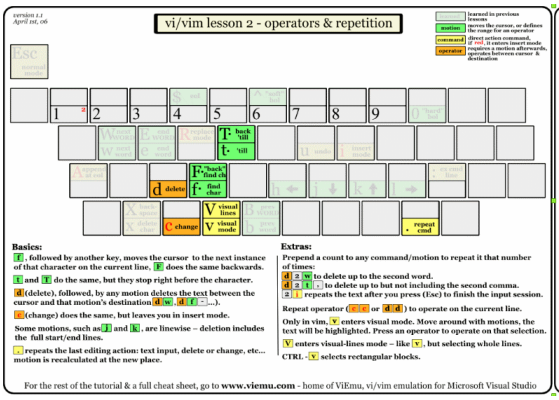
正像學習任一門編程語言一樣，學習用 VimL 進行插件開發，更重要的也是實踐。只不過 vim 一貫追求個性化，具體的插件開發可能沒那麼強的通用性，因而不適合作爲本書的正文內容。或許，那應該是另一個故事。而對讀者來說，那也纔算正式的起航。

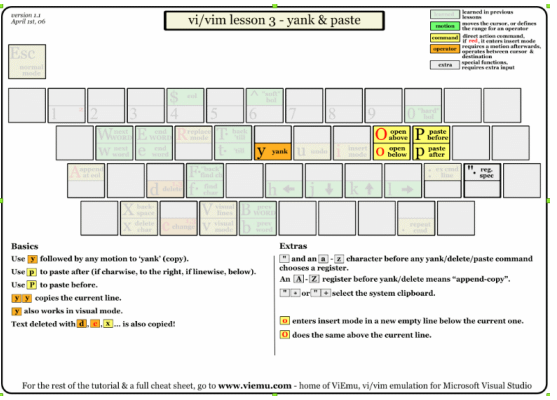
附錄

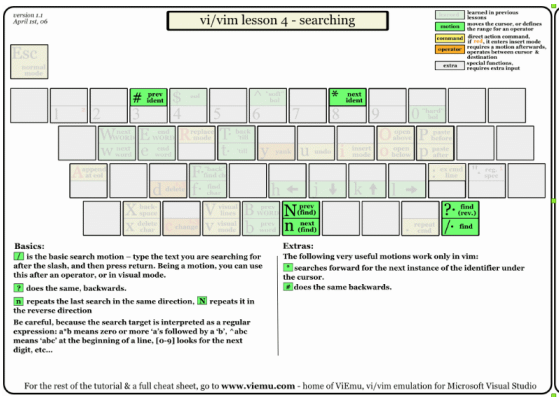
vi/vim 鍵位圖

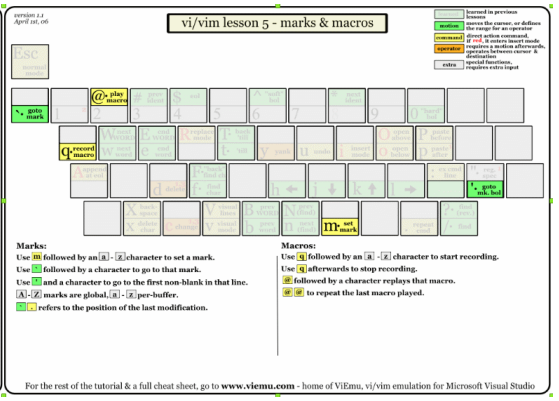
vi 的功能非常多，所以按鍵多，下面 lesson1-7 是基礎到進階的鍵位圖。

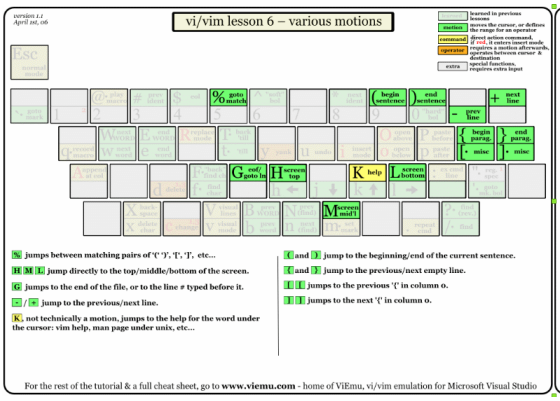


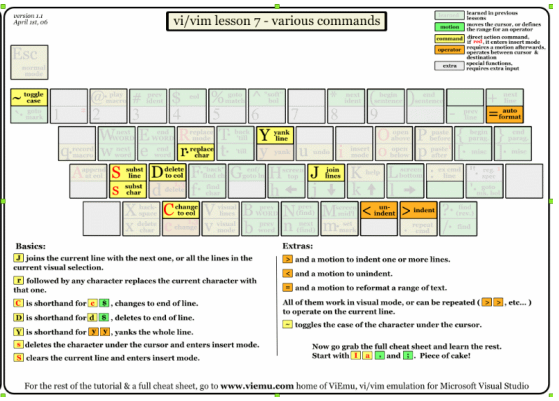




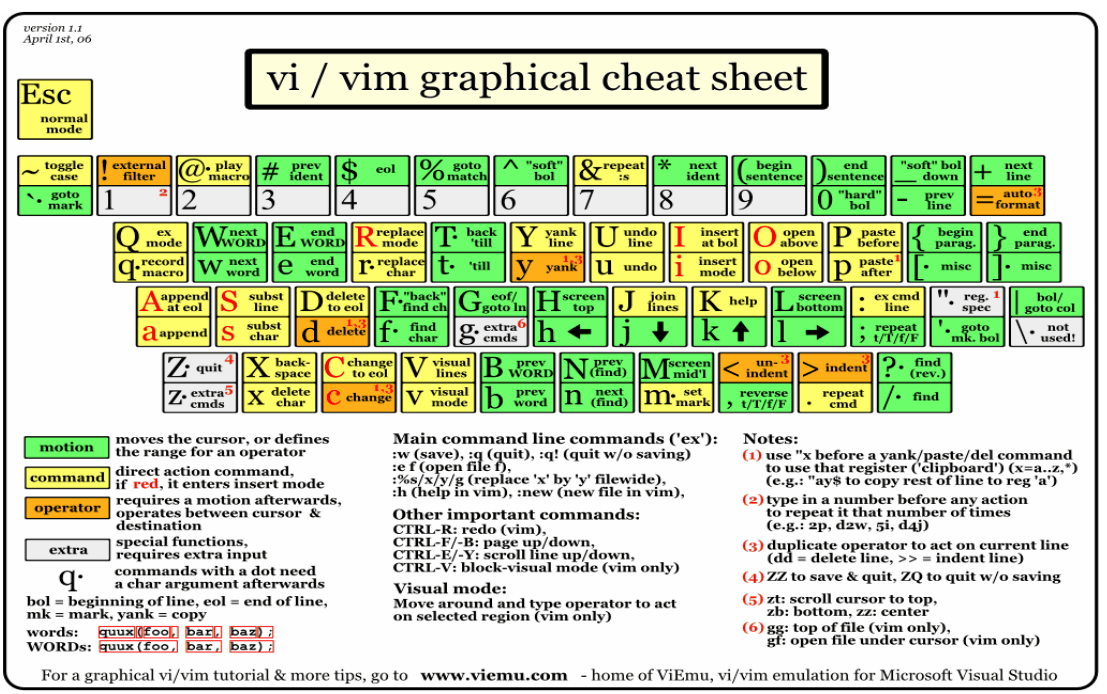








綜合一下，可以得到如下這幅更加詳細的鍵位圖。



vi 常用命令

進入 vi

vi filename	打開或新建文件，將光標置於第一行首
vi +n filename	打開文件，將光標置於第 n行首
vi + filename	打開文件，將光標置於最後一行首
vi -r filename	在上次正用 vi編輯時發生系統崩潰，恢復文件
vi file1....filen	打開多個文件，依次編輯

vi 的工作模式

命令行模式：控制屏幕光標的移動，字符、字或行的去除，
移動拷貝某區段及進入插入模式下，或者到底行模式

插入模式：只有在 Insert mode 下可以做文字輸入，
按「ESC」鍵可回到命令行模式

底行模式：將文件保存或退出 vi，也可以設置編輯環境，
如尋找字符串、列出行號等

命令行模式

移動光標

k、j、h、l	功能等同於上、下、左、右箭頭鍵
Ctrl+b	在文件中向上移動一頁
Ctrl+f	在文件中向下移動一頁
ctrl+u	屏幕往後移動半頁
ctrl+d	屏幕往前移動半頁
H	將光標移到屏幕的最上行 (Highest)
nH	將光標移到屏幕的第 n 行
M	將光標移到屏幕的中間 (Middle)
L	將光標移到屏幕的最下行 (Lowest)
nL	將光標移到屏幕的倒數第 n 行
w	在指定行右移光標，到下一個字的開頭
e	在指定行右移光標，到一個字的末尾
b	在指定行左移光標，到前一個字的開頭

0	數字0，左移光標，到本行的開頭
G	光標移動到文章的最後
nG	光標移動到文章的第 n行
$	右移光標，到本行的末尾
^	移動光標，到本行的第一個非空字符

替換和刪除

rc	用 c 替換光標所指向的當前字符
nrc	用 c 替換光標所指向的前 n 個字符
x	去除光標所在位置後面的一個字符
nx	去除光標所在位置後面的 n 個字符
X	大寫的 X，去除光標所在位置前面的一個字符
nX	去除光標所在位置前面的 n 個字符
dd	去除光標所在行和空隙
ndd	從光標所在行開始去除 n 行內容和空隙

複製和粘貼

從正文中刪除的內容（如字符、字或行）並沒有真正丟失，而是被剪切並複製到了一個內存緩衝區中。用戶可將其粘貼到正文中的指定位置。注意，這個緩衝區和操作系統自帶的緩衝區不同，比如你用 Ctrl+C 複製的內容不會影響到在 vi 裏複製的內容。

p	小寫字母 p，將緩衝區的內容粘貼到光標的後面
P	大寫字母 P，將緩衝區的內容粘貼到光標的前面

如果緩衝區的內容是字符或字，直接粘貼在光標的前面或後面；如果緩衝區的內容爲整行正文則粘貼在當前光標所在行的上一行或下一行。有時需要複製一段正文到新位置，同時保留原有位置的內容。這種情況下，首先應當把指定內容複製（而不是剪切）到內存緩衝區。完成這一操作的命令是：

yy	拷貝當前行到內存緩衝區
nyy	拷貝 n 行內容到內存緩衝區

搜索字符串

/str1	正向搜索字符串 str1
n	繼續搜索，找出 str1 字符串下次出現的位置
?str2	反向搜索字符串 str2

撤銷和重複

u	撤消前一條命令的結果
.	重复最後一條修改正文的命令

文本選中

v	字符選中命令
V	行選中命令

插入模式

進入和退出插入模式

i	在光標左側輸入正文
a	在光標右側輸入正文
o	在光標所在行的下一行增添新行
O	在光標所在行的上一行增添新行
I	在光標所在行的開頭輸入正文
A	在光標所在行的末尾輸入正文
ESC/ Ctrl + [	退出插入模式

底行模式

在 vi 的底行模式下，可以使用複雜的命令。

退出命令

在命令模式下可以用 ZZ 命令退出 vi 編輯程序，該命令保存對正文所作的修改，覆蓋原始文件。如果只需要退出編輯程序，而不打算保存編輯的內容，可用下面的命令：

:q	在未作修改的情況下退出
:q!	放弃所有修改，退出編輯程序

行號和文件保存

:n	將光標移到第 n 行
:set nu	顯示行號
:set nonu	取消行號顯示

底行模式下，可以規定命令操作的行號範圍。數值用來指定絕對行號；字符“.”表示光標所在行的行號；字符“$”表示正文最後一行的行號；簡單的表達式，例如“.+5”表示當前行往下的第 5 行。例如：

```
: .+5              將光標移到當前行之後的第5行
: $                將光標移到正文最後一行
```

底行模式下，允許從文件中讀取正文，或將正文寫入文件。例如：

```
:w                 將內容寫入原文件
:wq                將內容寫入始文件，退出編輯程序
:w file            將內容寫入 file 文件，原文件內容不變
:a,bw file         將第 a 行至第 b 行的內容寫入文件 file
:r file            讀取 file 文件，插入當前光標所在行的後面
:f file            將當前文件重命名爲 file
```

字符串搜索

```
:/str/             正向搜索，將光標移到下一個包含 str 的行
:?str?             反向搜索，將光標移到上一個包含 str 的行
```

正文替換

```
:s/str1/str2/       用 str2 替換行中首次出現的 str1
:s/str1/str2/g      用 str2 替換行中所有出現的 str1
:.,$ s/str1/str2/g 用 str2 替換當前行到末尾所有的 str1
:1,$ s/str1/str2/g 用 str2 替換正文中所有的 str1
:g/str1/s//str2/g   功能同上
```

從上述替換命令可以看到：g 放在命令末尾，表示對搜索字符串的每次出現進行替換；不加 g，表示只對搜索字符串的首次出現進行替換；g 放在命令開頭，表示對正文中所有包含搜索字符串的行進行替換操作。

刪除正文

```
:d                 去除光標所在行
:3d                去除第 3 行
:.,$d              去除當前行至正文的末尾
:/str1/,/str2/d    去除從字符串 str1 到 str2 的所有行
```

恢復文件

vi 在編輯某個文件時，會另外生成一個臨時文件，這個文件的名稱通常以. 開頭，並以.swp 結尾。vi 在正常退出時，該文件被刪除，若意外退出，而沒有保存文件的最新修改內容，則可以使用恢復命令 ":recover file"，也可以在啓動 vi 時利用 -r 選項。

選項設置

爲控制不同的編輯功能，vi 提供了很多內部選項。利用:set 命令可以設置選項。基本語法爲:

```
:set option          設置選項 option

常見的功能選項包括：
autoindent           設置該選項，則正文自動縮進
ignorecase           設置該選項，則忽略規則表達式中大小寫區別
number               設置該選項，則顯示正文行號
ruler                設置該選項，則在底部顯示所在行、列的位置
tabstop              設置按 Tab 鍵跳過的空格數
mk                   將選項保存在當前路徑的 .exrc 文件中
```

shell 切換

在編輯正文時, 利用 vi 底行模式下提供的 shell 切換命令, 無須退出 vi 即可執行 Linux 命令。

```
:!command            執行完shell 命令 command 後回到 vi
```

另外，在命令模式下，鍵入 K ，可命令 vi 查找光標所在單詞的手冊頁，相當於運行 man 命令。