《VimL 语言编程指北路》目录

前言

这是一篇有关 Vim 脚本语言的入门与进阶教程。是“指北”，不是“指南”，所以如果不慎指错了路，切勿见怪。不过要相信地球是圆的，绕了一圈之后，希望还是能找对目标的。

初学者如果第一章看不懂，建议直接看第二章；如果第二章看不懂，建议直接看第三章；如果第三章也看不懂，建议直接放弃治疗，汝须先培养对 vim 的信仰习惯。

以下……开始严肃话题。

正名约定

Vim 是上古神器之一，且能历久弥新，与时俱进。随着 Vim 的发展，Vim 的脚本也逐渐 发展壮大，支持的功能与特性越来越丰富，俨然成为一种新的或老的实用脚本语言。然而这语言的名字，网上的称谓似乎还有点五花八门。

为了行文统一与方便，在这里我采用“VimL”(Vim Language 缩写)来表示该脚本语言，用“Vim” 表示编辑器；而小写的 “vim” 则是指系统中可执行的编辑器程序，若从“VimL”角度看，也算是它的解释器程序；然后“vim script”就是存放“VimL”代码且可用“vim”解释运行它的文本文件。

目标假设

本教程针对的目标群体，假定是有使用 Vim 的基础及一定的编程基础。尽管我尽量从基本概念讲起，但一些最基础的东西怕无从再细致了。然后最重要的是要热爱 Vim，并且 有折腾的精神来打造或调教自己的 Vim。

其实，不管是使用 Vim 还是 VimL，最好的资源都是 Vim 的内置帮助文档（`:help`）。外部教程都不免有所侧重，较适于学习阶段的引领者。

本教程依据的 Vim 版本是 8.1，系统环境 Linux。但除了一些新特性，应该也适用 Vim7 以下版本。同时由于 Vim 本身是跨平台的，VimL 自然也与操作系统无关。虽然无法一一验证，但在一些重要的差异处，尽量在文中指出。

VimL 的优缺点

作为一种语言，首先指出 VimL 的缺点一是只能在 Vim 环境下运行，二是运行速度有点慢。但是，对于热衷 Vim 的程序猿，每天的编码工作都在 Vim 环境下，VimL 的编程方 式与 Vim 的操作方式无间密合，应该算是个优势。

另外，程序的运行速度都是相对的。所有的动态脚本语言，相对静态的编译语言，都很慢。但这不要紧，只要完成大部分工作能足够快，脚本的简单便捷性就能体现出来了。VimL 同样具有脚本语言这个共性。

用 Vim 编写 VimL 代码，另有个天然的优势，就是编辑器，解释器，与文档手册一体化，同时仍然保持了 Vim 的小巧，不像静态语言的IDE那么笨重。

编程思想基本是独立于语言的，大多数语言都是相通的。现代的高级脚本语言更是几乎都能提供差不多的功能。（而且，据说只要是“图灵完备”的语言，理论上都能做任何事）。所以，经常使用 Vim 的程序猿，如果想多学一门脚本语言，那 VimL 是个不坏的选择。

文本约定

本教程拟用 `.md` 文件书写，章用一级标题，节用二级标题，每节至少一个文件。初稿不一定严格按目录大纲的顺序书写，并且在此过程中或有增删调整。

带星号 `*` 的章节，表示略有艰深晦涩的内容，可以选择性略过。

关于示例代码块，`:` 开始的行表示 Vim 的命令行（也叫 ex 命令），`$` 开始的行表示从 shell 运行的命令行。较短的示例代码，可以直接输入或粘贴入 vim 的命令行，较 长的示例代码，建议保存 `.vim` 文件，然后 `:source`。

本书正文共十章，可粗略分为三部分。第 1-3 章为基础篇，第 4-7 章为中级篇，第 8-10 为高级篇。在行文组织上尽量循序渐进，建议按顺序阅读。文中经常用提示用 `:help` 命令查阅相关帮助主题，此后忘记细节时可随时查询。

第一章 VimL 语言主要特点

1.1 Hello World 的四种写法

按惯例，我们讨论一门语言，首先看下如何写最简单的 “Hello World” （程序）。由于 Vim 是高度自由的，VimL 也有多种不同的方式玩转 “Hello World”。

速观派：直接操起命令行

最快速的办法是在 Vim 命令行下用 `:echo` 命令输出 “Hello World”：

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

唯一需要注意的是，得把 “Hello World” 用引号括起来，单引号或双引号都可以。这样再按下回车就能在 Vim 的消息区显示出 “Hello World” 这行字符串了。

由于这条消息字符串很简短，一行就能显示完整，Vim 将其直接显示在命令行的位置，并且运行完直接返回 Vim 普通模式。如果字符串很长或多行字符串，则消息区将向上滚动，以显示完整的消息，用户需要额外按个回车才回普通模式。

试试在命令行输入这条命令，看看有啥不同反应：

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

好了，你已经学会了如何用 VimL 输出 “Hello World” 了。这也算编程吗？别逗了！其实，别把编程想得那么严肃，那这就算编程！

正规派：建立脚本文件

把刚才在命令行输入的那条命令保存在一个 `.vim` 后缀的文本文件中，那就是一个 `vim script` 了，这是用 VimL 编程比较正常的用法。

为了方便，建议在本地建个目录，用于保存本教程的示例代码。比如：

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

这将在 ~/.vim 目录下新建一个 vimllearn 目录，并用 vim 开始编辑一个名为 hello1.vim 的文件。vim 会为该文件新建一个缓冲区 buffer，在该 buffer 中输 入以下文本，然后输入命令 :w 保存：

```
"   hello1.vim
"   VimL hello world
"   lymslive
"   2017-08

echo 'Hello World!'

finish

      ~~
```

你其实可以只在该文件中写入 echo 'Hello World!' 这一行就够了。几点说明：1. 前面以一个双引号 " 开始的行是注释，注释也可以写在行尾。 2. 在脚本文件中，echo 命令前不必加冒号 :，但是加上冒号也是允许的。 3. finish 表示直接结束脚本，在之后的语句都不再被 vim 解析；这是可选的，没有 遇到 finish 就会执行到文件最后一行。

当有了 *.vim 脚本文件，就可以在 vim 环境中用 :source 命令加载运行了：

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

需要将脚本文件的路径写在 source 命令之后作为参数。如果当前 vim 正常编辑 hello1.vim 这个文件，则可用 % 表示当前文件的路径：

```
: source %
```

折腾并解释了这许久，终于可以通过 source 一个 vim 脚本打印输出 “Hello World” 了，与此前的效果是一样一样的。当然了，用 VimL 写脚本肯定不能只满足于写 “Hello World” 吧，所以这才是标准用法。

此外 Vim 的命令是可以简写的，source 可简写为 so。当你在写一个 vim 脚本时想 快速验证执行该脚本时，可以只输入：

```
: so %
```

如果还想更省键，就定义一个快捷键映射吧，比如：

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

可以将这行定义加入你的 vimrc 中，不过最好是放在 ~/.vim/ftplugin/vim.vim 中 ，并加上局部参数，让它只影响 *.vim 文件：

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

测试派：进入 Ex 模式

直接在命令行用 :echo 查看一些东西其实很有用的，可以快速验证一些记不清楚的细节。比如你想确认下在 VimL 中字符 '0' 是不是与数字 0 相等，可以这样：

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

但如果要连续输入多条命令并查看结果，每次都要（从普通模式）先输入个冒号，不免有些麻烦。这时，Ex 模式就有用了。默认情况下（若未在 vimrc 被改键映射），在普通模式下用 Q 键进入 Ex 模式。例如，在 Ex 模式下尝试各种输出 “Hell World” 的写法，看看不同引号对结果的影响：

```
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
```

最后，按提示用 visual 或简写 vi 命令回到普通模式。

Vim 的 Ex 模式有点像 VimL 的交互式的解释器，不过语法完全一样（有些脚本语言的交互式解释器与执行脚本有些不同的优化），仍然要用 echo 显示变量的值。

*索隐派：从 shell 直接运行

如果只为了运行一个 vim script 脚本，也不一定要先启动 vim 再 source，直接在启动 vim 时指定特定参数也能办到。-e 参数表示以 Ex 模式启动 vim，-S 参数启 动后立即 source 一个脚本。因此，也可以用如下的命令来输出 “Hello World”：

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

这就相当于使用 vim 解释器来运行 hello.vim :q :vi ‘ 以正常方式继续使用 Vim。

vim 本身的命令行启动参数其实还支持很多功能，请查阅 :help starting。其中还有个特殊的参数是 -s，如果与 -e 联用，就启动静默的批处理模式，试试这个：

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

没有任何反应输出？因为 -s 使普通的 echo 提示无效，看不到任何提示！赶紧输入 q 回车退出 vim 回到 shell。因为如果不小心按了其他键，vim 可能就将其当作命令来处理了，而且不会有任何错误提示，这就会让大部分人陷入不知如何退出 vim 的恐慌 。

虽然 vim -e -s 不适合来输出 “Hello World”，但如果你的脚本不是用来做这种无聊的任务，这种模式还是有用的。比如批处理，在完全不必启动 vim 可视编辑的情况下，批量地对一个文件或多个文件执行编辑任务。可达到类似 sed 的效果。而且，在 vim 脚本写好的情况下，不仅可以按批处理模式执行，也可以在正常 Vim 可视编辑某个文件 时，遇到类似需求时，也可以再手动 :source 脚本处理。

小结

运行 vim 脚本的常规方法用 :source 命令，而且有很多情况下并不需要手动输入 :source 命令，在满足一定条件下，vim 会自动帮你 source 一些脚本。vim 的启动参 数 -S 其实也是执行 :source。

Vim 的命令行可以随时手动输入一些简短命令以验证某些语法功能，进入 Ex 模式则可 以连续手动输入命令并执行。Ex 模式虽然比较少用，但不该反感排斥，这对学用 VimL 还是大有裨益的，以后会讲到，VimL 的 debug 功能也是在 Ex 模式中的。

静默批处理 vim -e -s 本质上也是 Ex 模式，不过禁用或避免了交互的中断。属于黑科技，一般的 vim 用户可先不必深究。

*拓展阅读：Vim 与可视化

“可视化”是个相对的概念。现在说到可视化，似乎是指功能丰富的 IDE 那种，有很多辅助窗口展示各方面的信息，甚至有图形化来表示类层次关系与函数调用关系。还有传说中的唯一的中文编程语言“易语言”还支持图文框拖拖拽拽就能编写代码的东东……而 vim 这 种古董，只有编辑纯文本，似乎就该归属于“不可视”。

然而，让我们回顾洪荒时代，体验一下什么叫真正的“不可视”编辑。

在 Vi 都还没诞生的时代，有一个叫 ed 的行编辑器，它只能通过命令以行为单位去操作或编辑文本文件。它完全没有界面，无从知道当前编辑的是哪个文件，在哪行，当前行是什么内容，用户只能“记住”，或用命令查询。比如用 p 命令打印显示当前行（不过也可以在前面带上行地址打印多行，至今 vim 的大部分命令都可以带地址参数）。要编辑当前行，请用 a i 或 c 命令（vimer 有点眼熟吧），不过编辑完后也无从知晓结果是否正确，可能还需要再用 p 命令打印查看确证。

之后，有个 ex 编辑器，不过是对 ed 的命令进行了扩展，本质上仍是行编辑器。直到 vi 横空出世，那才叫“屏幕编辑器”。意思是可以全屏显示文件的许多行，移动光标实时修改某一可见行，修改结果即时显示……这才像我们现在可认知的编辑器了。

然后是 vim 对 vi 的扩展增强。事实上，vim 还不如 vi 的划时代意义，它的增强与此前 ex 对 ed 的增强是差不多的程度，基本上是平行扩展。

可视化程度不是越高越好。 vim 与 vi 都保留与继承了 ex 的命令，因为 ex 命令确实高效，当你能确知一个命令的运行结果，就没必要关注中间过程了。比如最平凡无奇但常用的 ex 命令就是 :s 全局替换命令。

VimL 语言就是基于 ex 命令的，再加上一些流程控制，就成了一种完整的脚本语言。如果说 vim 对 vi 有什么压倒性的里程碑意义，我觉得应是丰富完善了 VimL 语言，使得 vim 有了无穷的扩展与定制。利用 VimL 写的插件，既可以走增加可视化的方向，也可以不增加可视化而偏向自动化。依每人的性格习惯不同，可能会在 Vim 的可视 化与自动化之间找到适合自己的不同的平衡点。

第一章 VimL 语言主要特点

1.2 同源 ex 命令行

那么，VimL 到底是种什么样的语言。这里先说结论吧，VimL 就是富有程序流程控制的 ex 命令。用个式子来表示就是：

```
VimL = ex   +
```

VimL 源于 ex ，基于 ex，即使它后来加了很多功能，也始终兼容 ex 命令。

然则什么是 ex 命令，这不好准确定义，形象地说，就是可以在 Vim 底部命令行输入并执行的语句。什么是流程控制，这也不好定义呢，类比地说，就是像其他大多语言支持的选择、循环分支，还有函数，因为函数调用也是种流程跳转。

下面，还是用些例子来阐述。

第一个脚本：vimrc

为了说明 vimrc 先假设你没有 vimrc 。这可以通过以下参数启动 vim：

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

这样启动的 vim 不会加载任何配置文件，可以假装自己是个只会用裸装 vim 的萌新。同 时也保证以下示例中所遇命令没有被重映射，始终能产生相同的结果。

Vim 主要功能是要用来编辑一些东西的，所以我们需要一些语料文本。这也可以用 vim 的普通命令生成，请在普通模式下依次输入以下按键（命令）：

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

其中输入的按键不包括换行符，上面分几行显示，只为方便分清楚几个步骤的命令。

首先是个 a 命令，进入插件模式，输入字符串“Hello World!”，然后按 <ESC> 键 返回普通模式（这里<ESC>表示那个众所周知的特殊键，不是五个字符啦）。a 之前 的 20 是还在普通模式下输入的数字参数，（它不会显示在光标所在的当前行，而是临时显示在右下角，然而一般不必关注）这表示后来的命令重复执行多少次。所以结果是在当前行插入了 20 个 “Hello World!”，也就是新文件的第一行。

接着命令 yy 是复制当前行，99p 是再粘贴 99 行。于是总共得到 100 行 “Hello World!” ——满屏尽是 Hello World!，应该相当壮观。

最后的命令是用冒号 : 进入 ex 命令行，保存文件，<CR> 表示回车，ex 命令需要回车确认执行。

现在，我们已经在用 vim 编辑一个名为 helloworld.txt 的文件了。看着有点素是不是？可以用下面的 ex 命令设置行号选项：

```
: set number
```

如此就会在文本窗口左则增加几列特殊列，为文件中的每行编号，确认一下是不是恰好 100 行，用 G 普通命令翻到最后一行。

还有，是不是觉得每一行太长了，超过了窗口右边界。（如果你用的是超大显示屏，Vim 的窗口足够大还没超过，那么在一开始的示例中，把数字参数 20 调大吧）如果想让 vim 折行显示，则用如下命令设置选项：

```
: set wrap
```

可以看到，长行都折行显示了，但是行编号并没有改变。也就是说文件中仍是只有 100 行，只有太长的行，vim 自动分几行显示在屏幕窗口上了。

你可以继续输入些设置命令让 vim 的外观更好看些，或让其操作方式更贴心些。但是等等，这样通过冒号一行行输入实在是太低效，应该把它保存到一个 vim 脚本文件中。

按冒号进入命令行后，再按 <Ctrl-F> 将打开一个命令行窗口，里面记录着刚才输入的 ex 历史命令。这个命令窗口的设计用意是为了便于重复执行一条历史命令，或在某条历史命令的基础上小修后再执行。不过现在我们要做的是将刚才输入的两条命令保存到一个文件中，比如就叫 vimrc.vim，整个按键序列是：

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

解释一下：进入命令窗口后光标自动在最后一行，V 表示进入行选择模式，k 上移一 行，即选择了最后两行。在选择模式下按 : 进入命令行，会自动添加 '<, '>，这是 特殊的行地址标记法，表示选区，然后用 :w 命令将这两行写入 vimrc.vim 文件（ 注意当前目录应在 ~/.vim/vimllearn 中）。最后的 :q 命令只是退出命令窗口，但 vim 仍处于编辑 helloworld.txt 状态中。

你需要再输入一个 :q 退出 vim，然后用刚才保存的脚本当作启动配置文件重新打开文件，看看效果：

```
:q<CR>
$ vim -u vimrc.vim helloworld.txt
```

可见，重新打开 helloworld.txt 文件后也自动设置了行号与折行。你可以换这个参数启动 vim 对比下效果，确认是 vimrc.vim 的功效：

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

可以手动编辑 vimrc.vim 增加更多配置命令：

```
: e vimrc.vim
```

这样就切换到编辑 ‘vimrc.vim’ 状态了，里面已经有了两行，用普通命令 Go 在末尾 打开新行进入插入模式，加入如下两行（还可自行添加一些注释）：

```
: 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 后重新用 $ vim -u vimrc.vim helloworld.txt 参数启动 vim 打开文件观察效果。也可以在当 前的 vim 环境中重新加载 vimrc.vim 令其生效：

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

其中，:e # 或快捷键 <Ctrl-^> 表示切换到最近编辑的另一个文件，这里就是 helloworld.txt 文件啦，在这个文件上移动 j k 键，看看是否有什么不同体验了 。

不过，表演到些为止吧。这段演示示例主要想说明几点：

1. VimL 语言没什么神秘，把一些 ex 命令保存到文件中就是 vim 脚本了。
2. vimrc 配置文件是 vim 启动时执行的第一个脚本，也应是大多数 Vim 初学者编写的 第一个实用脚本。

关于 vim “命令” 这个名词，还有一点要区分。普通模式下的按键也叫“命令”，可称之为“普通命令”，但由于普通模式是 Vim 的主模式，所以“普通命令”也往往简称为“命令”了 。通过冒号开始输入而用回车结束输入的，叫 “ex 命令”，vim 脚本文件不外是记录 “ex 命令”集。（注：宏大多是记录普通命令）

默认的 vimrc 位置

正常使用 vim 时不会带 -u 启动参数，它会从默认的位置去找配置文件。这可以在 shell 中执行这个命令来查看：

```
$ vim --version
```

或者在任一已启动的 vim 中用这个 ex 命令 :version 也是一样的输出。在中间一段 应该有类似这样几行：

```
  vimrc  : "$VIM/vimrc"
  vimrc  : "$HOME/.vimrc"
   vimrc  : "~/.vim/vimrc"
  exrc  : "$HOME/.exrc"
efaults file: "$VIMRUNTIME/defaults.vim"
```

它告诉了我们 vim 搜索 vimrc 的位置与顺序。主要是这两个地方，~/.vimrc，~/.vim/vimrc。用户可用其中一个做为自定义配置，强烈建议用第二个 ~/.vim/vimrc。因为配置可能渐渐变得很复杂，将所有配置放在一个目录下管理会更方便。不过有些低版本的 vim 可能不支持 ~/.vim/vimrc 配置文件，在 unix/linux 系统下可将其软链接为 ~/.vimrc 即可。

需要注意的是，vimrc 是个特殊的脚本，习惯上没有 .vim 后缀。

如何配置 vimrc 属于使用 Vim 的知识（或经验）范畴，不是本 VimL 教程的重点。不过为了说明 VimL 的特点，也给出一个简单的示例框架如下：

```
" 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 配置包括选项设置，快捷键映射，插件加载等几部分，每部分都可能变得复杂起来，为方便管理，可以分别写在不同的 vim 脚本中，然后在主 vimrc 脚本 中用 `:source` 命令调用。这就涉及脚本路径全名问题了，若期望能跨平台，就可创建一个变量，根据运行平台设置不同路径，这就用到了 `:if` 分支命令了。

最后两行打印个欢迎词。你可以将自己的大名赋给变量 `$USER`。如果，你觉得这很傻很天真，可以移到 finish 之后就不生效了。

在 vimrc 中，选择分支可能很常见，根据不同环境加载合适的配置嘛。但循环就很少见了。因为 vim 向来还有个追求是小巧，启动快，那么你在启动脚本中写个循环是几个意思啊，万一写个死循环BUG还启不起来了。

流程控制语句也是 ex 命令

在 VimL 中，每一行都是 `ex` 。作为一门脚本语言，最常见的，创建变量要用 `:let` 命令，调用函数要用 `:call` 命令。初学者最易犯与迷惑的错误，就是忘了 `let` 或 ‘call’，裸用变量，裸调函数，比如：

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

用过其他语言的可能会觉得这很自然，但在 VimL 中是个错误，因为它要求第一个词是钦定的 `ex` ！正确的写法是：

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

从这个意义上讲，VimL 与 shell 脚本很类似的，把命令行语句保存到文件中就成了脚本。每一行都以可执行命令开始，后面的都算作该命令的参数。

在 VimL 中，`:if` `:for` `:while` 也当作扩展的 `ex` 处理，在 vim 脚本中，这些“关键词”前面，可以像 `:set` 一样加个可选的冒号。同时，也可以像其他 ex 命令 一样在无歧义时任意简写。比如：

- :endif 可简写为 en 或 end 或 endi 或 endif
- :endfor 可简写为 endfo
- :endfunction 可简写为 endf，后面补上 unction 任意前几个字符也可以。

这套缩写规则，就与替换命令 :substitute 简写为 :s，设置命令 :set 简写为 :se 一样一样的。但是，在写脚本时，强烈建议都写命令全称。命令简写只为在命令行中快速输入，而在脚本中只要输入一次，一劳永逸，就应以可读性为重了。

当有了这个意识，VimL 的一些奇怪语法约定，也就显得容易理解多了。比如：

- ex 命令以回车结束，所以 VimL 语句也按行结束，不要在末尾加分号，加了反而是语法错误，在 Vim 中每个符号都往往有奇葩意义。
- VimL 的续行符 \ 写在下一行的开始，其他一些语言是把 \ 写在上一行结束，表示转义掉换行符，合为一行。但在 VimL 中，每一行都需要一个命令，你可以把 \ 想象为一个特殊命令，意思是“合并到上一行”。
- 在 VimL 中，不推荐在一行写多个语句，要写也可以，把反斜杠 \ 扶正为竖线 | 表示语句分隔吧。这在 vim 下临时手输单行命令时可能较为常见，减少额外按回车与冒号。在很多键盘布局中，|（与\）恰好在回车键上面。

关键命令列表

Vim 的 ex 集是个很大的集合，比绝大多数的语言的关键字都多一个数量级。幸运的是，我们写 VimL 语言的脚本，并不需要掌握或记住这所有的命令，只要记住一些主要的关键命令就可以完成大部分需求了。

我从 VimL 语言的角度，按常用度与重要度并结合功能性将那些主要的命令分类如下，仅供参考：

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. 其他着重于编辑功能的命令

其中，第 5 类恰好是个分界线，之上的是形成 VimL 语言的关键命令，之下是作为 Vim 编辑器的重要命令。没有后面的编辑器命令，纯 VimL 语言也可以写脚本，作为一种（没有什么优势的）通用脚本而已；只有利用后面的编辑器命令，才可以调控 Vim。

后面的编辑器命令也可以单独使用，所以 Vim 高手也未必一定需要会 VimL。不过有了 VimL 语言命令的助力，那些编辑器命令可变得更高效与灵活。不过最后一大类纯编辑命令，可能较少出现在 VimL 语言脚本中。因为按 Vim 可视化编辑的理念，是需要使用者对这些编辑结果作出及时反馈的。同时，很多编辑命令也有相应的函数，在 VimL 中调用 函数，可能更显得像脚本语言。所以，VimL 中不仅有“库函数”的概念，还有“库命令”呢。

总之，语句与命令，是联结 VimL 与 Vim 的重要纽带。这是 VimL 语言的重要特点，也是初学者的一大疑难点。尤其是对有其他语言编程经验的，可能还需要一定的思维转换过程吧。

第一章 VimL 语言主要特点

1.3 弱类型强作用域

“弱类型”不是 VimL 的特点，是几乎所有脚本语言的特点。准确地说是变量无类型，但值有类型。创建变量时不必定义类型，直接赋值就行，也可以认为是变量临时获得了值的类型。关于 VimL 的变量与类型，将在下一章的基础语法中详解。

变量作用域是编程的另一个重要概念，也几乎每个语言都要管理的任务。这里说 VimL 具有“强作用域”的特点，是指它提供了一种简明的语法，让用户强调变量的作用域范围。

VimL 语言级的作用域 g: l: s: a:

变量作用域的意义是指该变量在什么范围内可见，可被操作（读值或赋值）。在 VimL 中，每个变量都可以加上一个冒号前缀，表示该变量的作用域。不过另有两条规则：

1. 在一些上下文环境中，可以省略作用域前缀，等效于加上了默认的作用域前缀
2. 有一些作用域前缀只在特定的上下文环境中使用。

从 VimL 语言角色看，主要有以下几种作用域：

1. `g:` 全局作用域。全局变量就是在当前 vim 会话环境中，在任何脚本，任何 ex 命令行中都可以引用的变量。所有在函数之外的命令语句，都默认是全局变量。
2. `l:` 局部作用域。只可在当前执行的函数体内使用的变量，在函数体内的变量默认为局部变量，`l:`局部变量也只能在函数体内使用。
3. `s:` 脚本作用域。只有当前脚本内可引用的变量，包括该脚本的函数体内。
4. `a:` 参数作用域。特指函数的参数，在函数体内，要引用传入的实参，就得加上 `a:` 前缀，但定义函数时的形参，不能加 `a:` 前缀。`a:` 还隐含一个限定是只读性，即不能在函数体内不能修改参数。

这几种作用域前缀所对应的英文单词，可认为是 `global`, `local`, `script` 与 `argument`。不过 `s:` 也可认为是 `static`，因为在 C 语言中，`static` 也表示只在当前文件中有效的意思。

变量作用域的应用原则：

1. 尽量少用全局变量，因为容易混乱语用，难于管理。不过在 ex 命令行或 Ex 模式下 只为临时测试的语句，为了方便省略前缀，是全局变量，当然在此命令中也只能是全局变量。在写 vim 脚本文件时，若要使用全局变量，不要省略 `g:` 前缀。同时全局 变量名尽量取得特殊点，比如全是大写，或带个插件名的长变量名，以减少被冲突的概率。
2. 局部变量的前缀 `l:` 一般可省略。但我建议也始终加上，虽然多敲了两个字符，但编程的效率来源于思路清晰，不在于多少那几个字符。同时在 VimL 编程时，坚持习惯了作用域前缀，就能在头脑中无形地加强这种意识，然后对作用域的把握也更加精准。另外，显然地，在函数体内要引用全局就是必须加上 `g:` 前缀。

3. 在写 vim 脚本时，函数外的代码，能用 s: 变量就尽量用 s: 变量。对于比较大的脚本变量（如字典），想对外分享，也宁可先定义为 s: 变量，再定义一个全局可访问的函数来返回这个脚本变量。
4. 参数变量，a:是语法强制要求，漏写了 a: 往往是个错误，（如果它没报错，恰好与同名局部变量冲突了，那是更糟糕与难以觉察的错误）也是初写 VimL 函数最容 易犯的语法错误。

Vim 实体作用域 b: w: t:

Vim 作为一个可视化的编辑器，给用户呈现的，能让用户交互地操作的实体对象主要有 buffer（缓冲文件），window（窗口），tabpage（标签页）。可以把它们想象为互有关系的容器：

- 缓冲对应着一个正在编辑中的文件，在不细究的情况下可认为与文件等同。（不过不一定对应着硬盘上的一个文件，比如新建的尚未保存的文件，以及一些特殊缓冲文件）缓冲也可认为是容纳着文件中所有文本行的容器，就像是简单的字符串列表了。
- 窗口是用于展示缓冲文件的一部分在屏幕上的容器。Vim 可编辑的文件很大，极有可能在一个屏幕窗口中无法显示文件的所有内容吧，所以窗口对应于缓冲文件还有个可视范围。一个窗口在一个时刻只能容纳一个缓冲文件，但在不同时刻可以对应不同的缓冲文件。
- 标签页是可以同时容纳不同的窗口的另一层更大的容器。原始的 Vi 没有标签页，标 签页是 Vim 的扩展功能。标签页极大增强 Vim 的可视范围，可认为窗口是平面的，再叠上标签页就是（伪）立体的了。
- 一个缓冲文件可以展示在不同的窗口或（与）标签页中。所有已展示在某个窗口（包括在其他标签页的窗口）的缓冲文件都是“已加载”状态，其他曾经被编辑过但当前不可见的缓冲文件则是“未加载”状态，不过 Vim 仍然记录着所有这些缓冲文件的列表。

然后，Vim 还有个“当前位置”的概念。也就是光标所在的位置，决定了哪个是“当前缓冲文件”，“当前窗口”与“当前标签页”。

有了这些概念，对 VimL 中的另外三个作用域前缀 b: w: t: 就容易理解了。其意即指一个变量与特定的缓冲文件、窗口或标签页相关联的，以 b: 举例说明。

- b:varname 表示在当前缓冲文件（实体对象）中存在一个名为 “varname” 的变量。
- VimL 语句在执行过程中，只能直接引用当前缓冲文件的 b: 变量，如果要引用其他缓冲文件的变量，要么先用其他命令将目标缓冲文件切换为当前编辑的缓冲文件，或者调用其他的内置函数来访问。
- 如果一个缓冲文件“消失”了，那么与之关联的所有 b: 变量也消失了。
- 窗口与标签页的“消失”能比较形象与容易地理解，关闭了就算消失了。但 Vim 内部对 缓冲的管理比较复杂，未必是从窗口上不见了就代表“消失”了。
- 不过在一般 VimL 编程中，可暂不必深究缓冲文件什么时候“消失”。只要记着一个 b: 变量必定与一个缓冲文件关联着，不同的缓冲文件使用相同的 b: 变量是安全的， 它们互不影响。

作用域前缀其实是个字典

以上介绍的各种作用前缀，不仅是种语法约定的标记，它们本身也是个变量，是可以容纳保存其他变量的字典类型变量。关于字典，在后续章节再详述。这里只能介绍几个演示示例来体会一下这种特性。

为了操作环境一致，也假设按上节的“裸装” Vim 启动：（不过其实不太影响，也不必太拘泥）

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

现在已用 vim 打开了一个 helloworld.txt 文件。在命令行输入以下 ex 命令：

```
: let x = 1
: echo x
: echo g:x
: echo g:.x
: echo g:['x']
: echo g:
```

你可以每次按冒号进入命令行逐行输入，也可以先进入 Ex 模式，连续输入这几行，效果是一样的（以后不再注明）。

首先用 :let 命令定义了一个 x 变量，命令行语句默认是全局变量。然后用 :echo 命令使用几种不同写法来引用读取这个变量的值，这几种写法都是等效的。最后将 g: 当作一个整体变量打印显示。它就是个全局字典，能里面包含了 x 键，值就是 1。（如果按正常的 vim 启动，你的 vimrc 以及各种插件可能会提供很多全局变量，那么 echo g: 的内容可能很多，不只 x 哟）

然后我们再写个脚本观察下 s: 变量。:e hello2.vim，输入以下内容并保存：

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

脚本写完了，在 ex 命令行输入以下几条测试下：

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

可见，: source % 命令能正常执行，脚本内的 :echo s: 打印出了该脚本内定义的所有 s: 脚本变量。但在命令行直接试图访问 s: 变量则报错了。

在脚本中也可以访问全局变量。可以自行尝试在 hello2.vim 中加入对刚才在命令行定义的 g:x 变量的访问。不过在实际的编程中，可千万别在脚本中依赖在命令行建立的全局变量。

然后再测试下 b: 变量，直接在命令行执行以下语句吧：

```
: 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
```

这里，`:e #` 表示切换编辑另一个文件。在实际工作中，或者用快捷键 `<Ctrl-^>` 更 方便，不过在本教程中，为说明方便，采用 ex 命令 `:e #` 。在本例中，vim 启动时 打开的文件是 `helloworld.txt`，后来又编辑了 `hello2.vim`；此时用 `:e #` 命令就 切回编辑 `helloworld.txt`了，再执行 `:e #` 就再次回 `hello2.vim` 中，这是轮换 的效果。

这个示例结果表明，在编辑 `hello2.vim` 时定义了一个 `b:x` 变量，这与全局的的 `x` 变量是互不冲突的。但是在换到编辑 `helloworld.txt` 时，`b:x` 变量就不存在了，因为并未在该缓冲文件中定义 `b:x` 变量呀。重新回到编辑 `hello2.vim` 文件时，`b:x` 变量又能访问了。这也说明当缓冲文件“不可见”时，vim 内部管理它的对象实体其实并未“消失”呢。而全局变量 `g:x` 或 `x` 是始终能访问的。

最后要指出的是，局部作用域 `l:` 与参数作用域 `a:` 不能像 `s:` 或 `b:` 这样当作整体的字典变量，是两个例外。VimL 这样处理的原因，可能一是没必要，二是没效率。函数体内的局部作用域与参数作用域，太窄，没必要将局部变量另外保存一个字典；而且有效时间太短，函数在栈上反复重建销毁，额外维护一个字典没有明显好处就不浪费了。另外若要表示所有函数参数另有一个语法变量 `a:000` 可实现其功能。

其他特殊变量前缀 `$` `v:` `&`

这几个符号其实并不是作用域标记。不过既然也是变量前缀，也就一道说明一下，也好加以区分。

含 `$` 前缀的变量是环境变量。除了 vim 在启动时会从父进程（如 shell）自动继承一些环境变量，这些变量在使用上与全局变量没什么区别。不过要谨慎使用，一般建议只读，不要随便修改，没必要的话也不要随便创建多余的环境变量。（实际上环境变量与全局变量的最大区另是环境变量在 ex 命令中会自动展开为当前的具体值，比如可直接使用 `:e $MYVIMRC` 编辑启动加载的 `vimrc` 文件。但在 VimL 脚本中将环境变量当作全局变量使用完全没问题）

含 `v:` 前缀的变量是 vim 内部提供的预定义常量或变量。用户不能增删这类特殊变量，也不能修改其类型与含义。比如 `v:true` 与 `v:false` 分别用于表示逻辑值“真”与“假”。Vim 所支持的这类 `v:` 变量往往随着版本功能的增加而增加。从与时俱进的角度讲，vim 脚本中鼓励使用这类变量增加程序的可读性，但若想兼容低版本，还是考虑慎用。要检查当前 vim 版本是否支持某个 `v:` 变量，只要用 `:help` 命令查阅一下即可。

而且 v: 本身也是个字典集合变量，可用 :echo v: 命令查看所有这类变量。

含 & 前缀的变量表示选项的值，相当于把选项变量化，以便于在 VimL 中编程。所支持的选项集，也是由 Vim 版本决定的，用户当然无法定义与使用不存在的选项。这部分内容在后面讲选项设置时再行讨论。

第一章 VimL 语言主要特点

1.4* 自动加载脚本机制

前文已提及，vim 脚本主要用 :source 命令加载，然而很多情况下又不需要手动执行该命令。只要将脚本放在特定的目录下，vim 就有个机制能自动搜寻并加载。

Vim 插件搜索目录

首先要知道有 &runtimepath （常简写为 &rtp）这个选项。它与系统的环境变量 $PATH 有点类似，就是一组有序的目录名称，用于 Vim 在许多不同情况下搜寻 *.vim 脚本文件的。你可以在命令行输入 :echo &rtp 查看当前运行的 vim 有哪些“运行时目 录”，一般都会包含 ~/.vim 这个目录。

- 除了 vim 启动时的第一个配置文件 vimrc，运行时需要加载的脚本，一般都是从 &rtp 目录列表中搜索的。
- vim 启动时，会在所有 &rtp 目录下的 plugin/ 搜索 *.vim 文件，并加载所有找到的脚本文件。需要注意的是在 plugin/ 子目录下的所有脚本也会自动加载。除非你先在 vimrc 中用选项禁用加载插件这个行为。
- 当一个文件类型 &filetype 被识别时，Vim 会从所有 &rtp 目录下的 ftplugin/ 子目录中搜索以文件类型开始的脚本文件，然后加载执行。比如编辑一个 cpp 文件 时，ftplugin/ 目录下的 cpp.vim cpp_*.vim cpp/*.vim 都会被加载。

所以，我们自己写的脚本，如果想让它在 vim 启动时自动生效，就扔到 ~/.vim/plugin/ 目录下，想只针对某种文件类型生效，就扔到 ~/.vim/ftplugin/ 目录下。

目前主流的第三方插件，也会遵循这种子目录规范，然后安装时一般会将整个目录添加到 &rpt 中，以便让 Vim 找到对应的脚本。

VimL 的自动加载函数（延时加载）

Vim 一直有个追求的目标是启动快。当插件越来越多时，vim 启动时要解析大量的脚本文件，就会被拖慢了。这时就出现了一个 autoload 自动加载函数的机制，这个巧妙的方法可算是 VimL 发展的一个里程碑吧。而在这之前，须由用户在 plugin/*.vim 的复杂 脚本中用极具巧妙的编程技巧，才好实现延时加载。

虽然还没有讲到 VimL 的函数，但也可以在这里解释自动加载函数的原理与过程，毕竟这不需要涉及到函数的具体实现。

例如，有一个 ~/.vim/autoload/foo.vim 脚本（或在其他任一个 &rtp 目录下的 autoload/ 子目录也行），该脚本内定义一个函数 foo#bar()，其中 # 之前的部分必须与脚本文件名 foo.vim 相同。将有以下故事发生：

- 在 vim 启动时，完全不会读取 foo.vim 文件，也不知道它里面可能定义了什么复杂的脚本内容。
- 当 foo#bar() 第一次被调用时，比如从命令行中执行 :call foo#bar()，vim 发现 foo#bar 这个函数未定义，就会试图从这个函数名分析出它可能定义于 foo.vim 文件中。然后就从 &rtp 目录列表中，依次寻找其中 autoload/ 子目录的 foo.vim 文件。将所找到的第一个 foo.vim 脚本加载，并停止继续寻找。如果在所有 &rtp 目录下都找不到，那就是个错误了。
- 加载（即 :source）完 foo.vim ，再次响应 :call foo#bar() 的函数调用，就能正常执行了。
- 如果 foo.vim 文件中其实并没有定义 foo#bar() 这个函数，比如手误把函数名写错了，写成了 foo#Bar()，则 vim 在二次尝试执行 :call foo#bar() 时依然报错说“函数未定义”。
- 如果此后再次调用 :call foo#bar()，由于文件已加载，该函数是已定义的了，vim 就不需要再次寻找 foo.vim 文件了，直接执行就是。
- 如果 foo.vim 文件中还定义了一个 foo#bar2() 函数，由于之前是加载整个文件，foo#bar2() 也是个已定义函数，也就可以直接调用的 :call foo#bar2()。
- 如果尝试调用一个 foo.vim 文件中根本不存在函数，如 :call foo#nobar()。即使之前已经加载过 foo.vim 一次，由于这个 foo#nobar 函数未定义，vim 会再次从 &rtp 目录找到这个 foo.vim 文件再加载一次，然后再尝试 :call foo#nobar() 依然出错报错。

各种细节过程可能很复杂，但总体思想还是很简单，就是延时加载，只要在必要时才额外加载脚本。从用户使用角度，只要注意几点：

- 函数名 foo#bar() 必须与文件名 foo.vim 完全一致（大小写也最好一致）。如果脚本是在 autoload 的深层子目录下，那函数名也必须是相对于 autoload 的路径名，把路径分隔符 / 替换为 # 就是。即在 autoload/path/to/foo.vim 文件中定义的函数名应该是 path#to#foo#bar()。
- 从使用便利性上，一般是会定义快捷键或命令来调用 # 函数，并在首次使用时触发相关脚本的加载。
- # 函数是全局作用域的，也可以认为各层 # 是完整的命名空间，当然从任何地方访问时都须使用路径全名，即使从相同的脚本内访问也须用全名。
- 全局变量也可以用 # 命名，如 g:path#to#foo#varname 也能触发相应脚本文件的自动（延时）加载，不过一般没有函数应用那么广泛。
- 尽量将复杂业务逻辑代码写在 # 自动加载函数中，有时要注意不同 &rtp 目录下同名文件的屏蔽效应。

利用 VimL 的这个自动加载机制，还有效地避免了全局变量（函数）名的冲突问题，因为函数名包含了路径名，而一般文件系统下是不会有重名文件的。唯一的问题是，这个函数名有点长。

第二章 VimL 语言基本语法

2.1 变量与类型

VimL 语言的变量规则与其他大多数语言一样，可以（只允许）由字母、数字与下划线组成，且不能以数字开头。特殊之处在于还可以在变量名之前添加可选的作用域前缀，如

`g: l: s: b: w: t:`（`a:`又有点特殊，在定义函数参数时不要前缀，而在使用参数时需要前缀），这在第一章有专门讨论，此不再叙说。

VimL 所支持的变量（值）类型可由帮助（`:help type()`）查看。其中最主要最常用的有数字（number）、字符串（string）、列表（list）与字典（dictionary）四种，或者可以再进一步归纳为三种，因为前两种（数字与字符串）在绝大数情况下自动转换，在使用时几乎不必考虑其类型差别，只须知道它表示“一个值”，所以也称作标量。而列表与字典变量则是“多个值”的集合，所不同在于获取其中某个值的索引方式不同。

标量：数字与字符串

数字是（number）直译，其实就是其他大多数语言所称的整数（int）。数字是有符号的，包括正负数与 0 ，其取值范围与 Vim 的编译版本有关。经笔者的测试，vim8.0 支持 8 字节的有符号整数，低版本只支持 4 字节的有符号整数。数字经常（或常规功能）是用于命令的地址参数表示行号，或命令的重复次数，一般情况下不必考虑数字溢界的问题。当你需要用到 VimL 来表达很大的整数时，才要小心这个潜在的问题。

字符串也简单，用单引号或双引号括起来即可，它们的语义是完全一致的。不过有以下使用建议原则：

- 一般使用单引号表示字符串，如 `'string'`，毕竟双引号还可用于行末注释，尽量避免混淆。
- 如果需要使用转义如 `\n \t`，则使用双引号，单引号不支持转义。
- 如果字符串包含一种引号，则使用另一种引号括起整个字符串。
- 如果有包含一层引用，则内外层用不同的引号。

数字变量支持常见的数学运算（加减乘除模，`+-*/%`），字符串只支持连接运算符，用点号（`.`）表示。此外，一些内建函数与命令也会要求其参数是数字或字符串。也就是数字与字符串有不同的使用环境，VimL 便能依据上下文环境将数字或字符串进行自动转换，规则如下：

- 数字转字符串表示是显而易见的，就是十进制数的10个数字字符表示法；
- 字符串转数字时，只截前面像数字的子串，若不以数字字符开头，则转为数字 0 。

请测试：

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

需要特别注意的是字符串 `'1.23'` 只会自动转为字数 `1` ，而不是浮点数 `1.23`。

在 VimL 中输入数字常量时，也支持按二进制、八进制、十六进制的表示法，不过自动转字符串时只按十进制转换。请自行观察以下结果，如果不懂其他进制可无视。

```
: echo 0xff
: echo 020
```

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

列表：有序集合

列表，在其他语言中也有的叫数组，就是许多值的有序集合。VimL 的列表有以下要点：

- 创建列表语法，中括号整体，逗号分隔元素： `:let list = [0, 1, 2, 3]`
- 用中括号加数字索引访问元素：`:echo list[1]`
- 索引从 0 开始，支持负索引，-1 表示最后一个元素，访问不存在索引时报错。
- 索引也可以用整数变量。
- 不限长度，在需要时会自动扩展容量。

在列表创建后，用内建函数 `add()` 与 `remove()` 动态增删元素，用 `len()` 函数取得列表长度（元素个数）。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：无序集合

字典，在其他语言中可能叫 Hash 表或散列表，就是许多“键-值”对的集合。与列表最大的不同在于，它不是用数字索引来访问其内的元素，而是用字符串索引（键）来访问元素。字典在内部存储方式是无序的，但通过键访问元素的效率极快。

定义与使用字典的语法示例如下：

```
: 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
```

语法要点：

- 字典用大括号 `{}` 括号整体。
- 每个键值对用逗号分分隔，键与值用冒号分隔，键一般有引号表字符串。
- 大括号内的空白是可选的，最后一个逗号也是可选的。
- 访问字典内某个元素时，仍是中括号 `[]` 索引，键放在中括号中。

- 在创建字典或访问元素时，键既可用引号引起的常量字符串，也可用字符串变量，数字变量自动转换为字符串。
- 当一个键是普通常量字符串（可用作变量名的字符串）时，可不用中括号加引号索引，而简洁地用点号索引，二者等价。
- 不能访问不存在的键，否则报错。
- 能直接对不存在的键赋值，表示对字典增加一个键值对元素。

删除变量

创建（或叫定义）变量用 :let 命令，相应的也就有 :unlet 命令用于删除一个变量。一般情况下没必要删除一个标量，因为它也占不了多少内存，需要重定义时也可以重新赋值。但对于列表与字典，有时比较在意其集合意义，可以用 :unlet 删除其中一个值，加上对应的索引即可，如果不加索引，则表示删除整个列表或字典。

例如：假设 list 与 'dict' 变量已如上定义：

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

如果要删除的变量或字典（列表）不存在的索引，:unlet 会报错。如果想绕过该错误检测，则可用 :unlet! 命令。

在 vim8.0 版本之前，标量、列表、字典三者是不互通的。如果 list 已被定义成了一个列表变量，那么它就不能用 :let 重赋值为一个字典或字符串或其他什么，但允许重赋值为另一个列表变量。如果一定要改变 list 的变量类型，只能先 :unlet 它，再 重新 :let 它为其他任意变量。

在 vim8.0 版本之后，不再有这个限制，不会再报诸如“类型不匹配”的错误了，更好地体现了动态弱类型的特点。然而，良好的命名规范要求变量名望文生义，在同一个范围的同一个变量名，前后用之于表达完全不同类型的变量，并不是个好习惯。

浮点数

虽然浮点数在 VimL 中用的比较少，但毕竟还是支持的。

- 浮点数也叫小数，支持科学记数法。
- 数字（整数）可自动转为浮点数。
- 浮点不能自动转为整数，也不能自动转为字符串。
- 整数运算结果仍是整数，浮点数运算结果仍是浮点数。
- 浮点数取整后仍是浮点数，不是整数。

请看以下示例：

```
: echo 1.23e3
: let int = 123 | let float = 1.23 | let str = 'string'
: echo str . int
: echo str . float |"
: echo str . 5 / 3
```

```
: echo str . 5 % 3
: echo str . 5 / 3.0 |"
: echo str . 5 % 3.0 |"
: echo round(5/3.0)
: echo round(5/3.0) == 2
: echo round(5/3.0) . str |"
```

最后一行语句说明，虽然一个浮点数取整后看似与一个整数相等，但它仍然不是整数，所以不能与字符串自动连接。

要将一个字符串“显式”转换为整数，可以与 0 相加；同理，要将整数“显式”转换为字符串，可与空串 "" 相连接。VimL 还提供了另一个内建函数 string() 将任意其他类型转换为可打印字符串。于是，想将一个浮点数圆整为“真正”的整数，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：行末注释可用一个双引号 " 开始，但建议用 |" 更有适用性。| 表示分隔语句，只是后面一个语句是只有注释的空语句。

类型判断

从 Vim8.0 开始，有一系列 vim 变量专门地用来表示各种变量（值）类型。比如 v:t_list 表示列表类型。如果要判断一个变量是否列表类型，可用以下三种写法中任何一种（但之前的低版本 Vim 只能用后两种）：

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

关于具有选择分支功能的 :if 语句，在下一节继续讲解。

第二章 VimL 语言基本语法

2.2 选择与比较

vim 在执行脚本时，一般是按顺序逐条语句执行的。但如果只能像流水帐地顺序执行，未免太无趣了，功能也弱爆了。本节介绍顺序结构之外的最普遍的选择分支结构，它可以根据某种条件有选择地执行或不执行语句块（一条或多条语句）。

在 VimL 中通过 :if 命令来表示条件选择，其基本语法结构是：

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

如果满足表达式 {expr}，或说其值为“真”，则执行其后至 :endif 之间的语句。貌似突然迸进了许多新概念，得先理一理。

表达式与语句

什么叫表达式？这可难说了。我只能先描叙下在 VimL 中什么是与什么不是表达式：

- 单独的变量就是表达式，常量也是表达式，选项值（&option）也是，但选项本身不是；
- 函数调用是表达式；
- 表达式有值，表达式之间的合法运算的结果也还是表达式。
- 但表达式不是可执行语句，它只是语句的一部分。

至于语句，在第一章也讲过。VimL 语句就是 Vim 的 ex 命令行。笼统地说，有时说到 ex 是指整个命令行，不过狭义地说，是指它第一个单词所指代的关键命令。于是，VimL 语言的大部分语句，可认为遵循以下范式：

```
VimL = ex +
```

为什么说大部分呢？因为我们已经很熟悉的赋值语句如 :let i=1 就不完全适合。在这里，:let 是个命令，1 是个表达式。但 = 只是依附于 :let 命令的特殊语义符号，它不是个表达式，也不是个运算符。变量 i 在被创建之前，也还算不上表达式。而 i=1 写在一起，或为了增加可读性加些空白 i = 1，它也不是表达式，因为它没有值，（并不能像 C 语言那样使用连等号赋值），下面这两个语句是非法的：

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

在 VimL 中的常用语句中，除了这个基础得有点平淡无奇的赋值语句，其他大多是 + 范式的。比如已经大量使用的 :echo 语句，以及上节介绍过的给列表添加一个元素的函数调用语句 :call add(list, item) 。

然而，其实也不必太拘泥于这些概念名词。理解就好。我们归纳出概念也不外是为了更地理解。

逻辑值与条件表达式

:if 命令（以及下一节要将要的 :while命令）后面的表达式，就是一个条件表达式。它期望这个表达式的值的类型是逻辑值，即 type() 的结果是 v:t_bool(=6) 的值 。如果值的类型不是逻辑值，则会自动将其他值转换为一个逻辑值。逻辑类型只有唯二的两个值，v:true 表示真，v:false 表示假。

所以关键在于 VimL 如何判定其他值是否有真假，什么是真，什么是假？其转换规则如何？这直接写代码测试一下吧：

```
: 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 |"
```

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

"
: 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：由于语句比较简单，就将 :if 与 :endif 直接写在一行了，用 | 分隔子语句。正常代码建议写在不同行上且缩进布局。

结果归纳于下：

- 数字 0 为假，其他正数或负数为真；
- 字符串先自动转为数字，转为 0 的话认为假，能转为其他数字认为真；
- 浮点数不能转为逻辑值，无法判断真假；
- 列表与字典也不能直接判断真假。

其实可进一步归纳为一句话，在 VimL 中，只能对整数值判断真假，0 是假的，其他都是真的，字符串先自动转为数字再判断真假。其他类型的值不能直接判断真假。（至 vim8.0 版本是此规则，后面是否会改就不得而知了）

然而，我们还是经常需要判断其他类型的值的某种状态。这时可以利用一个内建函数 empty() 来帮忙。它可以接收任何类型的一个参数，如果它是“空”的，就返回真（ v:true），否则返回假（‘v:false’）。在很大程度上，它可以代替直接使用 :if 的 条件表达式，只不过在值上恰好是逻辑取反；优点则是写法统一，适用于所有类型。

在上面这个例子中，可以都再次尝试把 :if 后面的表达式包在 empty() 的参数中执 行看看结果，或用 !empty() 取反判断，如：

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

综合建议：用 :if !empty({expr}) 代替 :if {expr}，避免逻辑烧脑，并且大部分情况下应该是你想要的。

比较运算符

两个整数进行相等性的比较，或大小性的比较，结果返回一个或真或假的逻辑值。整数支持的比较运算符包括：==, !=, >, >=, <, <=。

浮点数支持与整数相同的比较运算，但由于浮点误差，不建议用相等性判断。

字符串也支持与整数相同的那六个比较运算。虽然整数在直接 :if 命令中自动转为数字处理，但在比较运算中表现良好，就是按正常的编码序逐字符比较。不过有一点特别要注意的是，字符串比较结果受选项 &ignorecase 的影响，即有可能按忽略大小写的方式来比较字符串。比如，观察一下如下结果吧：

```
: set ignorecase
```

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

因此，为了使比较结果不受用户个人的 vimrc 配置 &ignorecase 的影响，VimL 另　外提供两套限定大小写规则的比较运算符。在以上比较运算符之后再加　#　符号就表示 强制按大小写敏感方式比较，后面加上 ? 符号就表示强制按大小写不敏感的方式比较。比如：

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

所以，强烈建议在进行字符串比较时，只用 ==# 或 ==? 系的比较运算符。当然由于弱类型，字符串变量与数字变量其实是不可分的，所以将 ==# 或 ==? 之类的运用于数字上比较，也是完全没有关系的。

此外，字符串除了相等性比较，还有匹配性比较，即用 =~ !~ 运算符判断一个字符串是否匹配另一个作为正则表达式的字符串。正则表达式是另一个高级话题，这里不再展开。当然，匹配运算符也有限定大小写是否敏感的衍生运算符，而且一般建议用 =~# 与 !~# 匹配，毕竟正则表达本身有表达大小写的能力。

对于列表与字典变量，可进行相等性比较，但不能进行大小性比较。如果两个列表或字典的对应元素都相等，则认为它们相等。此外，列表与字典还另外有个同例性比较运算符，is 或 isnot。注意，这两个是类似 == 的运算符号，不是关键词，虽然它们用英文单词来表示。同样地，也有 is# 与 isnot? 的衍生运算，不过这主要为了语法的统一整齐，其实 is# is? 与 is 的结果是一致的。同例性比较的具体含义涉及实例引用的概念，这留待后面的章节继续展开。

逻辑运算符

在 VimL 中的逻辑值所支持的或、且、非运算并无意外，分别用符号 || && ! 表示就是，而且也支持短路计算特性。

- 或 expr1 || expr2, 只要 expr1 或 expr2 其中一个是真，整个表达式就是真，两个都是假才是假。如果 expr1 已经是真的，expr2 不必计算就直接获得真的结果。
- 且　expr1 && expr2，只有两个表达式都是真，结果才是真。如果　expr1 是假，则 不必计算 expr2 就返回结果假。
- 非 !expr ，对表达式真假取反。

if 分支流程

在了解这些逻辑值判断之后，理解 :if 的选择分支语句就容易多了，其完整语法结构如下：

```
: if {expr}
:     {block_if}
: elseif {expr}
:     {block_elseif}
: elseif {expr}
```

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

- 首先执行的是 :if 后面的 {expr} 表达式，它可能只是个简单表达式，也可能是多个逻辑值的复合运算，或者是很多表达式运算后得到的一个数字结果或逻辑值。只要它最终能被解释为真，就执行其后的 {block_if} 语句块。
- 如果 :if 的表达式为假，则依次寻找下一个表达式为真的 :elseif 语句块。
- 最后如果没有真的 :if 与 :elseif 条件满足，就执行 :else 语句块。
- 只有 :if 与 :endif 关键命令是必须的，:elseif 与 :else 及其语句块是可选的。
- 在任一条件下，最多只有一个语句块被执行，然后流程跳转到 :endif 之后，结束整 个选择分流程。
- 如果没有 :else 语句块，则在没有任何一个条件满足时，就不会执行任何一个语句块。在有 :else 时，则至少会执行一个语句块。

注意：elseif 是直接将 else 与 if 这两个单词拼在一起的，中间没有空格，也没有缩写。在许多不同的语言中，else if 的写法可能是变化最多的。

在 VimL 中，目前也没有 switch case 的类似语句，如果要实现多分支，只能叠加 :elseif。

在非常简单的 if else endif 语句中，也可以用条件表达式 expr1 : expr2 ? expr3， 这类似于：

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

整个表达式的值是 expr2 或 expr3 的值。至于条件表达式是否可以嵌套，这个我也不知道，反正我不用，也不建议用。就是条件表达式本身，也只推荐在一些有限的场合用，不推荐大量使用。因为一开始以为简单的逻辑判断，也可能以后会被修改的复杂起来，仍然是用 :if 清晰一些。

然后推荐 :if 的一个特殊技法。VimL 并没有块注释，但是可以把多行语句嵌套放在 :if 0 ... :endif 之间，然后其内的语句就完全不会被执行了，甚至有不合 VimL 语 法的行也没事。然而仍然只建议这样“注释”合法的语句行，因为 :if 0 的潜意识是在 某个时刻可能需要将其改为 :if 1 以重新激活语句。这主要是用于更方便地切换测试某块语句的运行效果。

```
: if 0
:
: endif
: echo 'done'
```

*运算符优先级

本节为讲叙选择分支语句，也引申讲了不少有关语句、表达式、运算符的相关问题。落到实处就是各种运算符的使用了，这就需要特别注意运算符的优先级问题。在此并不打算罗列 VimL 的运算符优先级表，因为到这里可能还有些内容未覆盖到。而且运算符优先级的问题太过琐碎，只看一遍教程并无多大助益，需要经常查文档，并自行验证。可以通过这个命令 `:help expression-syntax` 查看表达式语法表，其中也基本是按运算符优先级从低到高排列的，请经常查阅。

虽然由于运算符优先级会引起一些自己意想不到的问题，但回避这类问题的办法也是很简单的，这里是一些建议：

- 首先按自己的理解去使用运算符，要相信大部分语言的设计都是人性化的，不会故意设些奇怪的违反常理的规则。
- 对于自己不确定优先级，或者发现运算结果不符合自己所想时，添加小括号组合，使表达式运算的次序明确化。
- 拆分复杂表达式，借助中间变量，写成多行语句，不要写过长的语句。

第二章 VimL 语言基本语法

2.3 循环与遍历

程序的比人肉强大的另一个特性就是可以任劳任怨地重复地做些单调无聊（或有聊）的工作。本节介绍在 VimL 语言中，如何控制程序，命令其循环地按规则干活。

遍历集合变量

首先介绍的是如何依次访问列表如字典内的所有元素，毕竟在 2.1 节介绍的索引方法只适于偶尔访问查看某个具体的元素。这里要用到的是`for ... in` 语法。

例如遍历列表：

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

在这个例子中，变量 `item` 每次获取 `list` 列表中的一个元素，直到取完所有元素。相当于在循环中，依次执行 `:let item=list[0]` `:let item=list[1]` … 等语句。 这个变量也可以叫做“循环变量”。遍历列表保证是有序的。

对于字典的 `for ... in` 语法略有不同，因为在字典内的每个元素是个键值对，不仅仅是值而已。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：字典内的元素是无序的。

可以单独遍历键，利用内建函数 keys() 从字典变量中构造出一个列表：

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

这里的输出结果应该与上例完全一致。

遍历字典键时，如有需要，也可以先对键排个序：

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

遍历字典还有个只遍历值的方式，不过这种方式用途应该不多：

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

总之，对于 :for var in list 语句结构，var 变量每次获取列表 list 内的一个值。 字典不是列表，所以要利用函数 items() keys() values() 等先从中构造出一个 临时数组。

固定次数循环

如果要循环执行某个语句至某个固定次数，依然可利用 for ... in 语法。只不过要利用 range() 函数构造一个计次列表。例如，以下语句输出 Hello World! 5 次：

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

这里，我们用一个极简的合变量，单下划线 _ 来作为循环变量，因为我们在循环体中根本用不着这个变量。不过这种用法并不常见，这里只说明可用 range() 实现计次循 环。

那么，range() 函数到底产生了怎样的一个列表呢，这可用如下的示例来测试：

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

可见，range(n) 产出一个含 n 个元素的列表，元素内容即是数字从 0 开始直到 n， 但不包含 n，用数学术语就叫做左闭右开。

其实，range() 函数不仅可以接收一个参数，还可以接收额外参数，不同个数的参数使得其产出意义相当不一样，可用以下示例来理解一下：

```
: echo range(10)       |" => [0, 10)
: echo range(1, 10)    |" => [1, 10]
: echo range(1, 10, 2) |" =>   1     2       10
```

```
: echo range(0, 10, 2) |" =>  0    2      10
```

利用 `range()` 函数的这个性质，也就可以写出不同需求的计次 `for ... in` 循环。

注：VimL 没有类似 C 语言的三段式循环 `for( ; ; )`。只有这个 `for ... in` 循环，在某些语言中也叫 `foreach` 循环。

不定次数循环

不定循环用 `:while` 语句实现，当条件满足时，一直循环，基本结构如：

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

用 `:while` 循环一个重要的注意点是须在循环前定义循环变量，并记得在循环体内更新循环变量。否则容易出现死循环，如果出现死循环，vim 没响应，一般可用 `Ctrl-C` 中 断脚本或命令执行。

如果 `:while` 条件在一开始就不满足，则 `:while` 循环一次也不执行。在 `:for ... in` 循环中，空列表也是允许的，那就也不执行循环体。

在某些情况下，死循环是设计需求，那就可用 `:while 1` 或 `:while v:true` 来实现 ，而 `for` 循环无法实现，因为构建一个无限大的列表是不现实的。

循环内控制

循环除了正常结束，还另外有两个命令改变循环的执行流程：

- `:break` 结束整个循环，流程跳转到 `:endfor` 或 `:endwhile` 之后。
- `:continue` 提前结束本次循环，开始下次循环，流程跳转到循环开始，对于 `:for` 循环来说，循环变量将获取下一个值，对于 `:while` 循环来说，会再次执行条件判断。
- 这两个命令一般要结合 `:if` 条件语句使用，在特定条件下才改变流程，否则没有太多实际意义。

举些例子：

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

这里只打印了前 5 个数，因为当 `i` 变量到达 5 时，直接 `break` 了。

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

在这里，`i % 2` 是求模运算，如果是奇数，余数为 `1` ，`:if` 条件满足后由于 `:continue` 直接开始下一次循环，`:echo i` 就被跳过，所以只会打印偶数。

在用 `:while` 循环时，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'
```

这原意是将上个打印偶数的 `:for` 循环改为 `:while` 循环，但是好像陷入了死循环，先 `<Ctrl-C>` 中止再来分析原因。那原因就是 `:continue` 语句跳过了 `:let i+=1` 的循环变量更新语句，使它陷在同一个循环中再也出不来了。

所以，如果你的 `:while` 是需要更新循环变量的，而且还用了 `:continue`，最好将更新语句放在所有 `:continue` 之前。不过就这个例子而言，若作些修改后，还要同时修改一些判断逻辑，才能实现原有意图。

*循环变量作用域与生存期

对于 `:while` 循环，循环变量是在循环体之外定义的，它的作用域无可厚非应与循环结构本身同级。但对于 `:for` 循环，其循环变量是在循环头语句定义的，（可见 `:let` 并不是唯一定义或创建变量的命令，`:for`也可以呢），那么在整个 `:for` 结构结束之 后，循环变量是否还存在，值是什么呢？

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

在这个例子中，为避免之前创建的变量 `i` 的影响，先调用 `:unlet` 删了它，然后执行一个循环，在循环结束查看这个变量的值。可见在循环结束后，循环变量仍然存在，且其值是 `:for` 列表中的最后一个元素。

那么空循环又会怎样呢？

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

这个示例执行到最后会报错，提示变量不存在。所以循环变量 i 并未创建。因此准确地说，循环变量是在第一次进入循环时被赋值而创建的，而空循环就没能执行到这步。

再看一下示例：

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

在这个例子中，只在循环体最后多加了一个语句，:unlet i 将循环变量删除了。这种写法在 Vim7 以前版本中很常见。因为列表中是可以保存不同类型的其他变量的，甚至包括另一个列表或字典。因此在后续循环中，循环变量将可能被重赋与完全不同类型的值，这在 Vim7 是一个“类型不匹配”的错误。所以在每次循环后将循环变量删除，能避免这个错误，使之适用性更广。在 Vim8 之后，这种情况不再视为错误，所以这个 :unlet 语句不是必要。只是在这里故意加回去，讨论一下循环变量作用域与生存期的问题。

运行这个示例，可见在循环打印了 10 个数字后，最后那条语句报错，变量 i 不存在。这也是可理解的，因为这个变量在每次循环中反复删除重建。在第 10 次循环结束后， 删除了 i ，但循环无法再进入第 11 次循环，也就 i 没有再重建，所以之后 i 就不存在了。

这里想说明的问题是，如果从安全性考虑，或对变量的作用域有洁癖的话，可以在循环体内 :unlet 删除循环变量。这样可避免循环变量在循环结束后的误用，尤其是循环中有 :break 时，退出循环时那个循环变量的最后的值是很不直观的，你最好不要依赖它去做什么事情（除非是有意设计并考虑清楚了）。不过这有个显然的代价是反复删除重建变量会消耗一些性能（别说 VimL 反正慢就不注重性能了，性能都是相对的）。

小结

VimL 只有两种循环，:for ... in 与 :while。语义语法简单明了，没有其他太多变种需要记忆负担，掌握起来其实应该不难。

第二章 VimL 语言基本语法

2.4 函数定义与使用

函数是可重复调用的一段程序单元。在用程序解决一个比较大的功能时，知道如何拆分多个小功能，尤其是多次用到的辅助小功能，并将它们独立为一个个函数，是编程的基本素养吧。

VimL 函数语法

在 VimL 中定义函数的语法结构如下：（另参考 :help :function）

```
function[!]  (  )

endfunction
```

在其他地方调用函数时一般用 :call 命令，这能触发目标函数的函数体开始执行，以产生它所设计的功效。如果要接收函数的返回值，则不宜用 :call 命令，可用 :echo 观察函数的返回结果，或者用 :let 定义一个变量保存函数的返回结果。实际上，函数调用是一个表达式，任何需要表达式的地方，都可植入函数调用。例如：

```
call  ( )
echo  ( )
let   =  ( )
```

注：这里为了阐述方便，除了关键命令，直接用中文名字描述了。因而不是有效代码，在每行的前面也就不加 : 了。

函数名

函数名的命令规则，除了要遵循普通变量的命令规则外，还有条特殊规定。如果函数是在全局作用域，则只能以大写字母开头。

因为 vim 内建的命令与函数都以小写字母开始，而且随着版本提升，增加新命令与函数也是司空见惯的事。所以为了方便避免用户自定义命令与函数的冲突，它规定了用户定义命令与函数时必须以大写字母开头。从可操作 Vim 的角度，函数与命令在很大程度上是有些相似功能的。当然，如果将 VimL 视为一种纯粹的脚本语言，那函数也可以做些与 Vim 无关的事情。

习惯上，脚本中全局变量时会加 g: 前缀，但全局函数一般不加 g: 前缀。全局函数是期望用户可以直接从命令行用 :call 命令调用的，因而省略 g: 前缀是有意义的。当然更常见的是将函数调用再重映射为自定义命令或快捷键。

除了接口需要定义在全局作用域的函数外，其他一些辅助与实现函数更适合定义为脚本作用域的函数，即以 s: 前缀的函数，此时函数名可不一定要求以大写字母开头。毕竟脚本作用域的函数，不可能与全局作用域的内建函数冲突了。

函数返回值

函数体内可以用 :return 返回一个值，如果没有 :return 语句，在函数结束后默认返回 0。请看以下示例：

```
: function! Foo()
:     echo 'I am in Foo()'
: endfunction
:
: let ret = Foo()
: echo ret
```

你可以将这段代码保存在一个 .vim 脚本文件中，然后用 :source 加载执行它。如果你也正在用 vim 读该文档，可以用 V 选择所有代码行再按 y 复制，然后在命令行执行 :@"，这是 Vim 的寄存器用法，这里不准备展开详述。如果你在用其他工具读文档，原则上也可以将代码复制粘贴至 vim 的命令行中执行，但从外部程序复制内容至 vim 有时会有点麻烦，可能还涉及你的 vimrc 配置。因此还是复制保存为 .vim 文件再 :source 比较通用。

这段示例代码执行后，会显示两行，第一行输出表示它进到了函数 Foo() 内执行了，第二行输出表明它的默认返回值是 0。这个默认返回值的设定，可以想像为错误码，当函数正常结束时，返回 0 是很正常的事。

当然，根据函数的设计需求，可以显式地返回任何表达式或值。例如：

```
: function! Foo()
:     return range(10)
: endfunction
:
: let ret = Foo()
: echo ret
```

执行此例将打印出一个列表，这个列表是由函数 Foo() 生成并返回的。

注意一个细节，这里的 :function! 命令必须加 ! 符号，因为它正在重定义原来存在的 Foo() 函数。如果没有 ! ，vim 会阻止你重定义覆盖原有的函数，这也是一种保护机制吧。用户加上 ! 后，就认为用户明白自己的行为就是期望重定义同名函数。

一般在写脚本时，在脚本内定义的函数，建议始终加上 ! 强制符号。因为你在调试时可能经常要改一点代码后重新加载脚本，若没有 ! 覆盖指令，则会出错。然后在脚本调试完毕后，函数定义已定稿的情况下，假使由于什么原因也重新加载了脚本，也不外是将函数重定义为与原来一样的函数而已，大部分情况下这不是问题。（最好是在正常使用脚本时，能避免脚本的重新加载，这需要一些技巧）

不过这需要注意的是，避免不同脚本定义相同的全局函数名。

函数参数

在函数定义时可以在参数表中加入若干参数，然后在调用时也须使用相同数量的参数：

```
: function! Sum(x, y)
:     return a:x + a:y
: endfunction

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

在本例中定义了一个简单的求和函数，接收两个参数；然后调用者也传入两个参数，运行结果毫无惊喜地得到了结果 5 。

这里必须要指出的是，在函数体内使用参数 x 时，必须加上参数作用域前缀 a:，即 用 a:x 才是参数中的 x 形参变量。a:x 与函数之外的 x 变量（实则是 g:x ）毫无关系，如果在函数内也创建了个 x 变量（实则是 l:x），a:x与之也无关系 ，他们三者是互不冲突相扰的变量。

参数还有个特性，就是在函数体内是只读的，不能被重新赋值。其实由于函数传参是按值传递的。比如在上例中，调用 Sum(x, y) 时，是把 g:x 与 g:y 的值分别拷贝给 参数 a:x 与 a:y ，你即使能对 a:x a:y 作修改，也不会影响外面的 g:x g:y，函数调用结束后，这种修改毫无影响。然而，VimL 从语法上保证了参数不被修改 ，使形参始终保存着当前调用时实参的值，那是更加安全的做法。

为了更好地理解参数作用域，改写上面的代码如下：

```
: 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
```

在这个例子中，调用函数 Sum() 时，不再传入全局作用域的 x y 了，另外传入两 个常量，然后在函数体内查看各个作用域的 x 变量值。

结果表明，在函数体内，直接使用 x 代表的是 l:x，如果在函数内没定义局部变量 x，则使用 x 是个错误，它也不会扩展到全局作用域去取 g:x 的值。如果要在函 数内使用全局变量，必须指定 g: 前缀，同样要使用参数也必须使用 a: 前缀。

虽然在函数体内默认的变量作用域就是 l: ，但我还是建议在定义局部变量时显式地 写上 l:，就如定义 l:sum 这般。虽然略显麻烦，但语义更清晰，更像 VimL 的风格 。函数定义一般写在脚本文件，只用输入一次，多写两字符不多的。

至于脚本作用域变量，读者可自行将示例保存在文件中，然后也创建 s:x s:y 变量 试试。当然了，在正常的编程脚本中，请不要故意在不同作用域创建同名变量，以避免不必要的麻烦。（除非在某些特定情境下，按设计意图有必要用同名变量，那也始终注意加上作用域前缀加以区分）

函数属性：abort

VimL 在定义函数时，在参数表括号之后，还可以可选项指定几个属性。虽然在帮助文档 :help :function 中也称之为 argument，不过这与在调用时要传入的参数是完全不同的东西。所以在这我称之为函数属性。文档中称之为 argument 是指它作为 :function 这个 ex 的参数，就像我们要定义的函数名、参数表也是这个命令的 “参数”。

至 Vim8.0 ，函数支持以下几个特殊属性：

- abort，中断性，在函数体执行时，一旦发现错误，立即中断运行。
- range，范围性，函数可隐式地接收两个行地址参数。
- dict， 字典性，该函数必须通过字典键来调用。
- closure，闭包性，内嵌函数可作为闭包。

其中后面两个函数属性涉及相同高深的话题，留待第五章的函数进阶继续讨论。这里先只讨论前两个属性。

为理解 abort 属性，我们先来看一下，vim 在执行命令时，遇到错误会怎么办？

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

在这个例子中，第二行是个错误，因为 echo 要求表达式参数，但 error 这个词是 未定义变量。这里用 echomsg 代替 echo 是因为 echomsg 命令的输出会保存在 vim 的消息区，此后可以用 :message 命令重新查看；而 echo 只是临时查看。

将这几行语句写入一个临时脚本，比较 ~/.vim/vimllearn/cmd.vim ，然后用命令加载 :source ~/.vim/vimllearn/cmd.vim 。结果表明，虽然第二行报错了，但第三行仍然执行了。

不过，如果在 vim 下查看该文档，将这几行复制到寄存器中，再用 :@" 运行，第三行 语句就似乎不能被执行到了。然而这不是主流用法，可先不管这个差异。

然后，我们将错误语句放在一个函数中，看看怎样？

```
: function! Foo()
:     echomsg 'before error'
:     echomsg error
:     echomsg 'after error'
: endfunction
:
: echomsg 'before call Foo()'
: call Foo()
: echomsg 'after call Foo()'
```

将这个示例保存在 ~/.vim/vimllearn/t_abort1.vim，然后 :source 运行。结果错 误之后的语句也都将继续执行。

在函数定义行末加上 abort 参数，改为：

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

重新 :source 执行。结果表明，在函数体内错误之后的语句不再执行，但是调用这个出错函数之后的语句仍然执行。

现在你应该明白 abort 这个函数属性的意义了。一个良好习惯时，始终在定义函数时加上这个属性。因为一个函数我们期望它执行一件相对完整独立的工作，如果中间出错了，为何还有必要继续执行下去。立即终止这个函数，一方面便于跟踪调试，另一方面避免在错误的状态下继续执行可能造成的数据损失。

那为什么 vim 的默认行为是容忍错误呢？想想你的 vimrc，如果中间某行不慎出错了，如果直接终止运行脚本，那你的初始配置可能加载很不全了。Vim 在最初提供函数功能，可能也只是作为简单的命令包装重用，所以延续了这种默认行为。但是当 VimL 的函数 功能可以写得越来越复杂时，为了安全性与调试，立即终止的 abort 行为就很有必要 的。

如果你写的什么函数，确实有必要利用容忍错误这个默认特性，当然你可以选择不加 abort 这个属性。不过最好还是重新想想你的函数设计，如果真有这需求，是否直接写在脚本中而不要写在函数中更合适些。

*函数属性：range

函数的 range 属性，表明它很好地继承了 Vim 风格，因为很多命令之前都支持带行地址（或数字）参数的。不过 range 只影响一些特定功能的函数与函数使用方式，而在其他情况下，有没有 range 属性影响似乎都不大。

首先，只有在用 :call Fun() 调用函数时，在 :call 之前有行地址（也叫行范围）参数时，Fun() 函数的 range 属性才有可能影响。

那么，什么又是行地址参数呢。举个例子，你在 Vim 普通模式下按 V 进入选择模式，选了几行之后，按冒号 :，然后输入 call Fun()。你会发现，在选择模式下按冒号进入 ex 命令行时，vim 会自动在命令行加上 '<,'>。所以你实际将要运行的命令是 :'<,'>call Fun()。'< 与 '> 是两个特殊的 mark 位置，分别表示最近选区的第一行与最后一行。你也可以手动输入地址参数，比如 1,5call Fun() 或 1,$call Fun()，其中 $ 是个特殊地址，表示最后一行，当前行用 . 表示，还支持 + 与 - 表示相对当前行的相对地址。

总之，当用带行地址参数的 :{range}call 命令调用函数时，其含义是要在这些行范围内调用一个函数。如果该函数恰好指定了 range 属性，那么就会隐式地额外传两个参数给这个函数，a:firstline 表示第一行，a:lastline 表示最后一行。

比如若用 :1,5call Fun() 调用已指定 range 属性的函数 Fun()，那么在 Fun() 函数体内就能直接使用 a:firstline 与 ‘a:lastline’ 这两个参数了，其值分别为 1 与 5。如果用 :'<,'>call Fun() 调用，vim 也会自动从标记中计算出实际数字地址来传给 a:firstline 与 ‘a:lastline’ 参数。函数调用结束后，光标回到指定范围的第 1 行，也就是 a:firstline 那行。

如果用 :1,5call Fun() 调用时，Fun() 却没指定 range 属性时。那又该怎办，Fun() 函数内没有 a:firstline 与 a:lastline 参数来接收地址啊？此时，vim 会采用另一种策略，在指定的行范围内的每一行调一次目标函数。按这个实例，vim

会调 用 5 次 Fun() 函数，每次调用时分别将当前光标置于 1 至 5 行，如此在 Fun() 函数内就可直接操作 “当前行” 了。整个调用结束后，光标停留在范围内的最后一行。

函数的 range 属性的工作原理就是这样，然则它有什么用呢？如果函数在操作 vim 中 的当前 buffer 是极有用的。举个例子：

```
" File: ~/.vim/vimllearn/frange.vim

function! NumberLine() abort
    let l:sLine = getline('.')
    let l:sLine = line('.') . ' ' . l:sLine
    call setline('.', l:sLine)
endfunction

function! NumberLine2() abort range
    for l:line in range(a:firstline, a:lastline)
        let l:sLine = getline(l:line)
        let l:sLine = l:line . ' ' . l:sLine
        call setline(l:line, l:sLine)
    endfor
endfunction

finish
```

在这个脚本中，定义了一个 NumberLine() 不带 range 属性的函数，与一个带 range 属性的 NumberLine2() 函数。它们的功能差不多，就是给当前 buffer 内的 行编号，类似 set number 效果，只不过把行号写在文本行之前。

这里用到的几个内建函数稍作解释下，getline() 与 setline() 分别表示获取与设定文本行，它们的第一个参数都是行号，当前行号用 '.'表示。 line('.') 也表示获取 当前行号。

如果你正用 vim 编辑这个脚本，直接用 :source % 加载脚本，然后将光标移到 finish 之后，选定几行，按冒号进入命令行，调用 :'<,'>call NumberLine() 或 :'<,'>call NumberLine2() 看看效果。可用 u 撤销修改。然后可将光标移到其他地方，手动输入数字行号代替自动添加的 '<,'> 试试看。

最后，关于使用 range 属性的几点建议：

- 如果函数实现的功能，不涉及读取或修改当前 buffer 的文本行，完全不用管 range 属性。但在调用函数时，也请避免在 :call 之前加行地址参数，那样既无意义，还导致重复调用函数，影响效率。
- 如果函数功能就是要操作当前 buffer 的文本行，则根据自己的需求决定是否添加

range 属性。有这属性时，函数只调用一次，效率高些，但要自己编码控制行号，略复杂些。

- 综合建议就是，如果你懂 range 就用，不懂就不用。

*函数命令

:function 命令不仅可用来（在脚本中）定义函数，也可以用来（在命令行中）查看函数，这个特性就如 :command :map 一样的设计。

- :function 不带参数，列出所有当前 vim 会话已定义的函数（包括参数）。
- :function {name} 带一个函数名参数，必须是已定义的函数全名，则打印出该函数的定义。由此可见，vim 似乎通过函数名保存了一份函数定义代码的拷贝。
- :function /{pattern} 不需要全名，按正则表达式搜索函数，因为不带参数的 :function 可能列出太多的函数，如此可用这个命令过滤一下，但是也只会打印函数头，不包括函数体的实现代码，即使只匹配了一个函数。
- :function {name}() 请不要在命令行中使用这种方式，在函数名之后再加小括号，因为这就是定义一个函数的语法！

*函数定义 snip

在实际写 vim 脚本中，函数应该是最常用的结构单元了。然后函数定义的细节还挺多，endfunction 这词也有点长（脚本中不建议缩写）。如果你用过 ultisnips 或其他 类似的 snip 插件，则可考虑将常用函数定义的写法归纳为一个 snip。

作为参考示例，我将 fs 定义为写 s: 的代码片断模板：

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

关于 ultisnips 这插件的用法，请参考：https://github.com/SirVer/ultisnips

小结

函数是构建复杂程序的基本单元，请一定要掌握。函数必须先定义，再调用，通过参数与返回值与调用者交互。本节只讲了 VimL 函数的基础部分，函数的进阶用法后面另有章节专门讨论。

第二章 VimL 语言基本语法

2.5* 异常处理

异常是编程中相对高级的话题，也是比较有争议的话题。本教程旨在 VimL ，不可能 展开去讨论异常机制。所以如果你不了解异常，也不用异常，那就可完全跳过这节了。

如果你了解异常，并且不反对用异常，那么这里只是告诉你，VimL 也提供了语法支持，可以让你在脚本中使用异常，其基本语法结构如下：

```
try

catch / 1/
      1
catch / 2/
      2
...
finally

endtry
```

大致流程是这样的：先执行 `try` 下面的尝试语句块，如果这过程中不出现错误，那就没 `catch` 什么事了，但是如果有 `finally`，其后的收尾语句块也会执行。麻烦在于如果尝试语句块中有错误发生，就会抛出一个错误。错误用字符串消息的形式，所以 `catch` 用正则表达式捕获。由于错误消息可能有本地化翻译，所以匹配错误号比较通用。如果 `catch` 没有参数，则捕获所有错误。一旦错误被某个 `catch` 正确匹配了，就执行其后的异常处理语句块，然后如果有 `finally` 的话，收尾语句块也会执行。

如果在 `try` 中出现了错误，既没有 `catch` 捕获，也没有 `finally` 善后，那它就向上层继续抛出这个错误。直到有地方处理了这个错误，如果一直没能处理该错误，就终止脚本运行。

除了 vim 执行脚本中自动检测错误抛出外，也有个命令 `:throw` 可手动抛出。比较常见的在 `catch` 的异常处理块中，只处理了部分工作后，用 `:throw` 重新抛出错误让后续机制继续处理。`:throw` 不带参数时重新抛出最近相同的错误，否则可带上参数抛出指定错误。

虽然 VimL 也提供了这个一套完整的异常处理机制，但一般情况下用得不多。大约有以下原因：

- 使用 VimL 希望简单，用上异常就似乎很复杂了。
- vim 脚本本身就很安全，只能在 vim 环境下运行，似乎干不了什么坏事。而且 vim 早 就有备份相关的配置，对编辑保存的文件都可以备份的。

所以，除非要写个比较大与复杂的插件，用异常可能在代码组织上更为简洁，提供更良好的用户接口。

第三章 Vim 常用命令

在第二章已经介绍了 VimL 语言的基本语法，理论上来说，就可以据此写出让 vim 解释 执行的合法脚本了。然而，能写什么的脚本呢？除了打印“Hello World!”，以及高级点的 用循环计算诸如“1+2+…+100”这样人家好像也能心算的题目外，还能干嘛呢？

所以，如果要让 vim 脚本真正有实用价值，还得掌握 vim 提供的内置命令，用以控制 Vim 或定制 Vim。本章就来介绍一些主要的、常用用的命令。

Vim 是个极高自由度的文本编辑软件，它在以下几个层级上给用户提供了自由度：

1. option 选项。预设了一个很庞大的选项集，用户可以按自己的喜好设置每个选项的值（当然很多选项也可以接受默认址而假装当它们不存在），这可能改变 Vim 的很多 基础表现与行为。
2. map （快捷键）映射。一个非常简单但非常强大的机制。用户可以根据自己的习惯来重新映射各种模式下不同按键（及按键序列）的解释意义。初入门的 Vimer 很容易 沉迷于折腾各种快捷键。
3. command 自定义命令。Vim 是基于 ex 命令的，然后允许又你自定义 Ex 命令。可 见这是比简单映射更灵活强大的利器，当然它的使用要求也比映射要高一些。
4. VimL 脚本。进一步将命令升级为脚本语言，据此开发插件，使得 Vim 的扩展性具 有无限可能。在 Vim 社区已经涌现了很多优秀插件，大多可以直接拿来用。当自己掌握了 VimL 语言后，也就可以自己写些插件来满足自己的特殊需求或癖好。

本教程虽是旨在 VimL 脚本语言，但还是有必要从简单的选项说起吧。

3.1 选项设置

选项分类与设置命令

设置选项的命令是 set。根据选项值的不同情况，可以将选项分为以下三类：

1. 不需要值的选项，或者说是 bool 型的开关切换状态的选项。这种选项有两个相对立的选项名，分别用命令 :set option 表示开启选项，:set nooption 表示关闭选 项。例如 :set number 是设置显示行号，:set nonumber 是设置不显示行号。
2. 选项有一个值。用命令 :set option=value 设定该类选项的值。选项值可以是数字或字符串，但字符串的值也不能加引号，就按字面字符串理解。也就是说，:set 后 面的参数，不是 VimL 的表达式，与 :let 命令有根本的不同。这个命令更像是 shell 设置变量的语法，= 前后也最好不要用空格。
3. 选项允许有多个值，值之间用逗号分隔。设置命令形如 :set option=val1,val2。此外还支持 += 增量与 -= 减量语法，如 :set option+=val3 或 :set option-=val2，表示在原来的“值集合”的基础上增加某个值或移除某个值。

选项值变量

在选项名前面加个 & 符号，就将一个选项变成了相应的选项值变量。例如，以下两条命令是等效的：

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

与普通变量赋一样，= 前后的空格是可选的，这里的空格只是一种编程习惯，为增加可读性。另外有以下几点要注意：

1. 第一类选项，在用 :set 命令时不需要等号，但是用 :let & 命令时也要用等号将其值赋为 1 或 0，分别表示开启选项与关闭选项。同时 & 只允许作用在没有 no 前缀的选项之前。比如 :let &nonumber = 1 是非法的，只能用 :let &number = 0 表示相同意图。

2. 第二类选项，如果值是字符串，用 `:let &` 命令时要将值用引号括起来，也就像普通变量赋值一样，要求等号后面是合法的表达式。
3. 第三类选项，它的值也是一个由逗号分隔的（长）字符串，比如 `:echo &rtp`。并不 能由于这类选项支持多个值就将 VimL 的列表赋给它，不过很容易通过 `split()` 函 数从这类选项值中分隔出一个列表。

备注：选项设置 `:set` 应是历史渊源最早的命令之一吧。而 `:let` 是后来 VimL 语言 发展丰富起来提供的命令。两者有不一样的语法，所以又提供了这种等价转换方法。

vimrc 配置全局选项

严格地说，`:set` 是设置全局选项的命令。既是影响全局的选项，一般是要第一时间在 `vimrc` 中配置的。最重要的是以下两条配置：

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

第一条配置是说不要兼容 `vi`，否则可能有很多 `vim` 的高级功能用不了。第二条配置（虽然不是 `set` 选项）是用 Vim 编写程序源代码必要的，意思是自动检测文件类型，加载插件，自动缩进的意思。除非在很老旧的机器上，或为了研究需要，一般都没理由不加上这两条至关重要的配置。

下面再介绍一些比较重要的几类配置选项，当然这远远不够全面。查看选项的帮助命令是 `:help options`，查看某一个选项的帮助是在用单引号括起选项名作为帮助参数，例如 `:help 'option'`。查看某个选项的当前值是命令 `:set option?` 或 `:echo &option` 。

- 编码相关：encoding fileencodings fileencoding
- encoding 是 Vim 内部使用的编码。建议 `:set encoding=utf-8`。
- fileencodings 是打开文件时，Vim 用于猜测检测文件编码的一个编码列表。对中文用户，建议 `:set fileencodings=ucs-bom,utf-8,gb18030,cp936,latin1`。
- fileencoding （局部选项），当前文件的编码，如果与 encoding 不同，在写入时自动转码。用户一般不必手动设这个选项，除非你想用另外一种编码保存文件。
- 外观相关：number/relativenumber wrap statusline/tabline
- number 是在窗口左侧加一列区域显示行号，relativenumber 显示相对行号，即相对光标所在的行的行号，当前行是 0，上面的行是负数，下面的行是正数。
- wrap 是指很长的文本行折行显示。一般良好风格的程序源文件不应出现长行，但 Vim 作为通用文件编辑器，不一定只用于编辑程序。
- statusline 是定制状态栏，格式比较复杂，建议查看文档，也有些插件提供了很炫酷的状态栏。tabline 的定制格式与状态栏一样，在开多个标签页时才生效。
- laststatus 什么时候显示状态栏，建议用值 2 表示始终显示状态栏。
- cmdheight 命令行的高度，默认只有 1 行太少，当命令行有输出时可能经常要多按一个回车才回到普通模式。建议 2 行，更多就浪费空间了。

- wildmenu 这是在编辑命令行时，按补全键后，会临时在状态栏位置显示补全提示。
- GUI外观：只在 gVim 或有 GUI 版本的 Vim 有效
- guioptions 设置 GUI 各部件（菜单工具栏滚动条等）是否显示。
- clipboard 设置剪切板与 Vim 的哪个寄存器关联。
- 颜色主题：
- colorscheme 这是个单独的命令，不是 **set** 选项。选择一个颜色主题。颜色主题是放在运行时各路径的 **colors/** 子目录的 ***.vim** 文件。
- background 背景是深色 **dark** 或浅色 **light**。有的 colorscheme 只适于深色或 浅色背景，有的则分别为不同背景色定义不同的颜色主题。
- term 与 t_Co 有的颜色主题可能还与终端与终端色数量有关。
- cursorline 与 cursorcolumn 用不格式高亮当前行与当前列，具体高亮格式由颜色主题定义。个人建议只高亮 cursorline 。
- hlsearch 高亮搜索结果。
- 格式控制：
- formatoptions 控制自动格式化文本的许多选项，建议看文档。
- textwidth 文本行宽度，超过该宽度（默认78）时自动加回车换回。在编辑程序源文件时可用上个 formatoptions 选项控制只在注释中自动换行但代码行不自动换行。
- autoindent smartindent 插入模式下回车自动缩进。
- shiftwidth 缩进宽度。
- tabstop softtabstop 制表符宽度，软制表符是行首按制符缩进的宽度。一般建议硬制表符宽度 tabstop 保持 8 不变，用 shiftwidth softtabstop 表示缩进。
- expandtab 插入制表符自动转为合适数量的空格。
- paste 将 Vim 的插入模式置于“粘贴”模式，从外部复制文本进 Vim 开启该选项可避 免一些副作用。但只建议临时开启该选项。
- 路径相关：
- runtimepath 运行时路径，简称（rtp）。vim 在运行时搜索脚本的一组路径。一般不手动设置该值，如果有插件管理器管理插件的话。插件必须放在某个 **&rtp** 路径 下，现在流行的是将插件工程主目录添加至 Vim 的 **&rtp** 中。
- packpath （Vim8开始才支持）动态加载插件的搜索路径，默认是 **~/.vim/pack**。插件主目录可置于 **{packpath}/{packname}/opt/{plugin}**。然后用 **:packadd** 启用插件。
- path 这是 vim 在寻找编辑文件，如 **gf** **:find** 等命令时所要搜索的一组目录。

- tags 这是 vim 按标签跳转 Ctrl-] 或 :tag 等命令所依据的标签文件，默认是 ./tags,tags（相对路径）。一般不建议修改默认值，但可以默认值基础上添加更多的标签文件，比如编辑一个工程时，将工程主目录下的 tags 文件也加进来。
- autochdir 将当前路径自动切换到当前编辑的文件所在的目录。当你依赖一些管理工程类的插件时，可能要求当前路径锁定在工程主目录，不宜开启该选项。但是个人喜欢开启这插件，这样在用 :e 命令打开同目录下的其他文件时很方便。

Vim 所支持的选项实在是太多了。初学者建议参考前人经验成熟的配置，用 :help 查 看每个选项的具体含义，然后决定这种选项是否适合自己。另外注意有些选项可能要配合起来才能发挥更好的效果。

VimL 控制局部选项

局部选项是只影响当前缓冲文件或窗口（buffer/window）的选项。严格来说是局部选项值，而不是有另外一类选项。默认情况下每个新文件或窗口都继承选项的全局值，但对于一些选项，可以为该文件或窗口设定一个不同与全局的局部值。然而并不是所有选项都有局部值意义，在每个选项的帮助文档中，会指明该选项是全局（global）或局部的（local to buffer 或 local to window）。

设置局部选项（值）用 :setlocal 命令。如果目标选项没有局部值，则等效 :set 命令设置全局值。但是最好不要混用，避免误解。局部选项值变量用 &l:option 表示 。比如 number 行号就是个局部选项：

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

你可以将 vim 分裂出两个窗口（:split 或 :vsplit），在其中一个窗口上执行以上语句，试试看结果。需要注意的是，虽然局部选项值借用了变量的局部作用域前缀 l: ，但它的默认规则又有点不同。看这里的 &number 是默认的 &l:number 而不是 &g:number。事实上，普通的局部变量 l:var 根本不能在函数外的命令行使用。

当用 VimL 写脚本时，如果要改变选项设置，且该选项支持局部值，最好用 :setlocal 只改变局部值。这也是编程的一大原则，尽量将影响局部化。下面介绍一些比较重要的局部选项设置：

- 文件类型：filetype
- 大部分情况下，这个选项不用手写设，也不用脚本显式地设，打开自动检测就可以自动根据后缀名设置相应的文件类型。不过在创建新的文件类型时，可能需要自己设置这个选项。
- 文件类型插件，如 ~/.vim/ftplugin/*.vim 脚本内若涉及选项更改，也尽量用 :setlocal 只设局部选项。
- 缓冲类型：buftype
- “buffer type” 与 “file type” 是两个不同的概念。缓冲类型更加抽象，是 vim 内 部用于管理缓冲一些控制属性，而文件类型是着眼于文件内容性质的。

- buftype 的两个重要的选项值是 nofile 与 nowrite，表示特殊的不用写文件的 buffer，而这两者又还有细微差别，具体请读文档。
- 其他 buffer 属性：
- buflisted 是否将当前缓冲记录在缓冲列表中。
- bufhidden 当缓冲不再任一窗口展示，如何处理该缓冲，有几种不同的选项值。
- modifiable 当前缓冲是否可修改，包括更改编码与换行符格式也算种修改。

由于在 Vim 中，最主要的可见（可编辑）对象就只是 buffer，所以在一些复杂而细致的插件中，经常会开辟一个辅助窗口，仅为展示辅助内容，这就往往要设置一个特殊的 buftype 及其他一些 buffer 属性。

此外，在脚本中，可能有需求只临时改变某个选项值，处理完毕后再恢复原选项设置，这就要借且选项值变量了。处理流程大致如下：

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

第三章 Vim 常用命令

3.2 快捷键重映射

几乎每个初窥门径的 vimer 都曾为它的键映射欣喜若狂吧，因为它定制起来实在是太简洁了，却又似能搞出无尽的花样。

快捷键，或称映射，在 Vim 文档中的术语叫 “map”，它的基本用法如下：

```
map {lhs} {rhs}
map
```

其中快捷键 {lhs} 不一定是单键，也可能是一个（较短的）按键序列，然后 vim 将其 解释为另一个（可能较长较复杂的）的按键序列 {rhs}。为方便叙述，我们将 {lhs} 称为“左参数”，而将 {rhs} 称为“右参数”。左参数是源序列，也可叫被映射键，右参数是目标序列，也可叫映射键。

例如，在 vim 的默认解释下，普通模式下大写的 Y 与两个小写的 yy 是完全相同的功能，就是复制当前行。如果你觉得这浪费了快捷键资源，可将 Y 重定义为复制当前行从当前光标列到列尾的部分，用下面这个映射命令就能实现：

```
: map Y y$
```

然而，映射虽然初看起来简单，其中涉及的门道还是很曲折的。让我们先回顾一下 Vim 的模式。

Vim 的主要模式

模式是 Vim 与其他大多数编辑器的一个显著区别。在不同的模式下，vim 对用户按键的 响应意义有根本的差别。Vim 支持很多种模式，但最主要的模式是以下几种：

- 普通模式，这是 Vim 的默认模式，在其他大多模式下按 <Esc> 键都将回到普通模式。在该模式下按键被解释为普通命令用以完成快速移动、查找、复制粘贴等操作。
- 插入模式，类似其他“正常”编辑的模式，键盘上的字母、数字、标点等可见符号当作直接的字符插入到当前缓冲文件中。从普通模式进入插件模式的命令有：aAiIoO
- a 在当前光标后面开始插入，
- i 在当前光标之前开始插入，
- A 在当前行末尾开始插入，
- I 在当前行末首开始插入，
- o 在当前行下面打开新的一行开始插入，
- o 在当前行上面打开新的一行开始插入。
- 可视模式（visual），非正式场合下也可称之为“选择”模式。在该模式下原来的移动命令变成改变选区。选区文本往往有不同的高亮模式，使用户更清楚地看到后续命令将要操作的目标文本区域。从普通模式下，有三个键分别进入三种不同的可视模式：
- v （小写 v）字符可视模式，可以按字符选择文本，
- V （大写 V）行可视模式，按行选择文本（jk有效，hl无效），
- Ctrl-v 列块可视模式，可选择不同行的相同一列如几列。（Vim 还另有一种“select” 模式，与可视模式的选择意义不同，按键输入直接覆盖替换所选择的文本）
- 命令行模式。就是在普通模式时按冒号 : 进入的模式，此时 Vim 窗口最后一行将变 成可编辑输入的命令行（独立于当前所编辑的缓冲文件），按回车执行该命令行后回到普通模式。 本教程所说的 VimL 语言其实不外也是可以在命令行中输入的语句。此外还有一种“Ex 模式”，与命令行模式类似，不过在回车执行完后仍停留在该模式，可继续输入执行命令，不必每次再输入冒号。在“Ex模式”下用 :vi 命令才回到普通模式。

大部分初、中级 Vim 用户只要掌握这四种模式就可以了。对应不同模式，就有不同的映射命令，表示所定义的快捷键只能用于相应的模式下：

- 普通模式：nmap
- 插入模式：imap
- 可视模式：vmap （三种不同可视模式并不区分，也包括选择模式）
- 命令模式：cmap

如果不指定模式，直接的 map 命令则同时可作用于普通模式与可视选择模式以及命令后缀模式（Operator-pending，后文单独讲）。而 map! 则同时作用于插入模式与命令行模式，即相当于 imap 与 cmap 的综合体。其实 vmap 也是 xmap（可视模式）与 smap （选择模式）的综合体，只是 smap 用得很少，vmap 更便于记忆（v 命令进入可视模式），因此我在定义可视选择模式下的快捷键时倾向于用 vmap。

在其他情况下，建议用对应模式的映射命令，也就是将模式简名作为 map 的限定前缀。而不建议用太过宽泛的 map 或 map! 命令。

特殊键表示

在 map 系列命令中，{lhs} 与 {rhs} 部分可直接表示一般字符，但若要映射（或被映射）的不可打印字符，则要特殊的标记（<>尖括号内不分大小写）：

- 空格：<Space> 。映射命令之后的各个参数要用空格分开，所以若正是要重定义空格键意义，就得用 <Space> 表示。同时映射命令尽量避免尾部空格，因为有些映射会

把尾部空格当作最后一个参数的一部分。始终用 <Space> 是安全可靠的。

- 竖线：<BAR>。| 在命令行中一般用于分隔多条语句，因此要重定义这个键要用 <BAR> 表示。
- 叹号：<Bang>。! 可用于很多命令之后，用以修饰该命令，使之做一些相关但不同的工作，相当于特殊的额外参数。映射中要用到这个符号最好也以 <Bang> 表示。
- 制表符：<Tab>，回车：<CR>
- 退格：<BS>，删除键： <DEL>，插入键： <Ins>
- 方向键：<UP> <DOWN> <LEFT> <RIGHT>
- 功能键：<F1> <F2> 等
- Ctrl 修饰键：<C-x> （这表示同时按下 Ctrl 键与 x 键）
- Shift 修饰键：<S->，对于一般字母，直接用大写字母表示即可，如 A 即可，不 必有<S-a>。一般对特殊键可双修饰键时才用到，如 <C-S-a>。
- Alt <A-> 或 Meta <M-> 修饰键。在 term 中运行的 vim 可能不方便映射这个修饰键。
- 小括号：<lt>，大括号 <gt>
- 直接用字符编码表示：<Char->，后面可接十进制或十六进制或八进制数字。如 <Char-0x7f> 表示编码为 127 那个字符。这种方法虽然统一，但如有可能，优先使用上述意义明确方便识记的特殊键名表示法。

此外，还有几个特殊标记并不是特指哪个可从键盘输入的按键： * <Leader> 代表 mapleader 这个变量的值，一般叫做快捷键前缀，默认是 \。同 时还有个 <LocalLeader>，它取的是 maplocalleader 的变量值，常用于局部映射 。 * <SID> 当映射命令用于脚本文件中（应该经常是这种情况），<SID> 用于指代当前脚本作用域的函数，故一般用于 {rhs} 部分。当 vim 执行映射命令时，实际会把 <SID> 替换为 <SNR>dd_ 样式，其中 dd 表示当前脚本编号，可用 :scriptnames 查看所有已加载的脚本，同时也列出每个脚本的编号。 * <Plug> 一种特殊标记，可以避免与用户能从键盘输入的任
中，表示该映射来自某插件。与 <SID> 关联某一特定脚本不同，<Plug> 并不关联特定插件的脚本文件。它的意义请继续看下一节。

键映射链的用途与陷阱

键映射是可传递的，例如若有以下映射命令：

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

当用户按下 x，vim 首先将其解释为相当于按下 y，然后发现 y 也被映射了，于是最终解释为相当于按下 z。

这就是键映射的传递链特性。那这有什么用呢，为什么不直接定义为 :map x z 呢？假 如 z 是个很复杂的按键命令，比如 LongZZZZZZZ，那么就可先为它定义一个简短的映射名，如 y：

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

然后再可以将其他多个键如 x1 与 x2 都映射为 y，不必重复多次写 LongZZZZZZZ 了。然而，这似乎仍然很无趣，真正有意义的是用于 <Plug>。

假设在某个插件文件中有如下映射命令：

```
: map <Plug>(do_some_funny_thing) :call <SID>ActualFunction()<CR>
: map x <Plug>(do_some_funny_thing)
: map <C-x> <Plug>(do_some_funny_thing)
: map <Leader>x <Plug>(do_some_funny_thing)
```

在第一个映射命令中，其 {lhs} 部分是 <Plug>(do_some_funny_thing)，这也是一个“按键序列”，不过第一键是 <Plug>（其实不可能从键盘输入的键），然后接一个左括号，接着是一串普通字符按键，最后还是个右括号。其中左右括号不是必须的，甚至可以不必配对，中间也不一定只能普通字符，加一些任意特殊字符也是允许的。不过当前许多优秀的插件作者都自觉遵守这个范式：<Plug>(mapping_name)。

该命令的 {rhs} 部分是 :call <SID>ActualFunction()<CR>，表示调用当前脚本中定义的一个函数，用以完成实际的工作。然而 <Plug>... 是不可能由用户按出来的键序列，所以需要再定义一个映射 :map x <Plug>...，让一个可以方便按出的键 x 来 触发这个特殊键序列 <Plug>...，并最终调用函数工作。当然了，在普通模式的下几乎每个普通字母 vim 都有特殊意义（不一定是 x，而x表示删除一个字符），你可能不应该重定义这个字母按键，可加上 <Leader> 前缀修饰或其他修饰键。

那么为何不直接定义 :map x :call <SID>ActualFunction()<CR> 呢？一是为了封装隐藏实现，二是可为映射取个易记的映射名如 <Plug>(mapping_name)。这样，插件作者只将 <Plug>(mapping_name) 暴露给用户，用户也可以自己按需要喜好重定义触发键映射，如 :map y <Plug>(mapping_name)。

因此，<Plug> 不过是某个普通按键序列的特殊前缀而已，特殊得让它不可能从键盘输入，主要只用于映射传递，同时该中间序列还可取个意义明确好记的名字。一些插件作者为了进一步避免这个中间序列被冲突的可能性，还在序列中加入插件名，比如改长为：<Plug>(plug_name_mapping_name)。

不过，映射传递链可能会引起另一个麻烦。例如请看如下这个映射：

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

在打开具有长文本行的文件时，如果开启了折行显示选项（&wrap），则 gj 或 gk 命令表示按屏幕行移动，这可能比按文件行的 j k 移动更方便。所以这两个键的重映射是有意义的，可惜残酷的事实是这并没有达到想要的效果。作了这两个映射命令之后，若试图按 j 或 k 时，vim 会报错，指出循环定义链太长了。因为 vim 试图作以下解释：

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

无尽循环了，当达到一些深度限制后，vim 就不干了。

为了避免这个问题，vim 提供了另一套命令，在 map 命令之前加上 nore 前缀改为 noremap 即可，表示不要对该命令的 {rhs} 部分再次解析映射了。

```
: noremap j gj
```

```
: noremap k gk
```

当然，前面还提到，良好的映射命令习惯是显示限定模式，模式前缀还应在 nore 前缀 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
```

结论就是：除了有意设计的 <Plug> 映射必须用 :map 命令外，其他映射尽量习惯用 :noremap 命令，以避免可能的循环映射的麻烦。例如对本节开始提出的示例规范改写如下：

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

其中，:<C-u> 并不是什么特殊语法，只不过表示当按下冒号刚进入行时先按个 <C-u>， 用以先清空当前命令行，确保在执行后面那个命令时不会被其他可能的命令行字符干扰。（比如若不用 nnoremap 而用 noremap 时，在可视模式选了一部分文本后，按冒号就会自己加成 :'<,'>，此时在命令行中先按 <C-u> 就能把前面的地址标记清除。在很小心地用了 nnoremap 时，还会不会有情况情况导致干扰字符呢，也不好说，反正加上 <C-u> 没坏处。但若你的函数本就设计为允许接收行地址参数，则最好额外定义 :vnoremap，不用 <C-u> 的版本。）

各种映射命令

前面讲了最基础的 :map 命令，还有更安全的 :noremap 命令，以及各种模式前缀限定的命令 :nnoremap :inoremap 等。这已经能组合出一大群映射命令了，不过它们仍只算是一类映射命令，就是定义映射的命令。此外，vim 还提供了其他几个映射相关的命令。

- 退化的映射定义命令用于列表查询。不带参数的 :map 裸命令会列出当前已重定义的所有映射。带一个参数的 :map {lhs} 会列出以 {lhs} 开头的映射。同样支持模式前缀缩小查询范围，但由于只为查询，没有 nore 中缀的必要。定义映射的命令，至少含 {lhs} 与 {rhs} 两个参数。
- 删除指定映射的命令 :unmap {lhs}，需要带一个完全匹配的左参数（不像查询命令只要求匹配开头，毕竟删除命令比较危险）。可以限定模式前缀，如 nunmap {lhs} 只删除普通模式下的映射 {lhs}。注意，模式前缀始终是在最前面，如果你把 un 也视为 map 命令的中缀的话。
- 清除所有映射的命令 :mapclear。因为清除所有，所以不需要参数了。当然也可限定模式前缀，如 :nmapclear，表示只清除普通模式下的映射。另外还可以有个 <buffer> 参数，表示只清除当前 buffer 内的局部映射。这类特殊参数在下节继续讲解。

特殊映射参数

映射命令支持许多特殊参数，也用 <> 括起来。但它们不同于特殊键标记，并不是左参数或右参数序列的一部分。同时必须紧跟映射命令之后，左参数 {lhs}

之前，并用 空格分隔参数。

- <buffer> 表示只影响当前 buffer 的映射，:map :unmap 与 :mapclear 都可 接收这个局部参数。
- <nowait> 字面意思是不再等待。较短的局部映射将掩盖较长的全局映射。

<nowait> 这个参数很少用到。但其中涉及到的一个映射机制有必要了解。假设有如下两个映射定义：

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

因为定义的是两个按键的序列，当用户按下 x 键时，vim 会等待一小段时间，以判断用户是否想用 x1 或 x2 快捷键，然后触发相应的映射定义。如果超过一定时间后用户没有按任何键，就按默认的 x 键意义处理了。当然如果后面接着的按键不匹配任何映射，也是按实际按键解释其意义。

因此，若还定义单键 x 的映射：

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

当用户想通过按 x 键来触发该映射时，由于 x1 与 x2 的存在，仍然需要等待一小段时间才能确定用户确实是想用 x 键来触发 simple-thing 这件事。这样的迟滞效应可不是个好体验。

于是就提出 <nowait> 参数，与 <buffer> 参数联用，可避免等待：

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

这样，在当前 buffer 中按下 x 键时就能直接做 local-thing 这件事了。

尽管有这个效用，但 <nowait> 在实践中还是用得很少。用户在自行设定快捷键时，最好还是遵循“相同前缀等长快捷键”的原则。也就说当定义 x1 或 x2 快捷键后，就最好不要再定义 x 或 x123 这样的变长快捷键了。规划整齐点，体验会好很多。当然，如实在想为某个功能定义更方便的快捷键快，可定义为重复按键 xx，因为重复按键的效率会比按不同键快一点。（想想 vim 内置的 dd 与 yy 命令）

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

另一方面，局部映射参数 <buffer> 却是非常常用，鼓励多用。局部映射会覆盖相同的全局映射，而且当 <nowait> 存在时，会进一步隐藏全局中更长的映射。

- <silent> 在默认情况下，当按下某个映射的 {lhs} 序列键中，vim 下面的命令行 会显示 {rhs} 序列键。加上这个 <silent> 参数时，就不会回显了。我的建议是一般没必要加这个参数禁用这个特性。当映射键正常工作时，你不必去理会它的回显，但是当映射键没按预想的工作时，你就可在回显中看到它实际映射成什么 {rhs} 了 ，这可帮助你判断是由于映射被覆盖了还是映射本身哪里写错了。
- <special> 这是相对过时的参数了，它指示当前这个映射命令中接受 <> 标记特殊 键。在默认不兼容 vi 的设置下，不必加这个参数也能直接用 <> 表示特殊键。
- <script> 当坚持用 :noremap 代替 :map 这个参数也没什么用了。它的本意是限定右参数 {rhs} 不会再与脚本外部的映射相互作用了。

- <unique> 唯一性要求是确保不会覆盖原来已定义的映射。在使用命令 :map <unique> {lhs} {rhs} 时，如果发现 {lhs} 在此前已定义，这条重定义映射的命令就会失败。

这个参数一般用在共享插件中，为了避免覆盖用户自己已定义的映射。不过在脚本中，还有两个函数能作更好的控制。内建函数 mapcheck() 用于判断一个 {lhs} 是否已被映射，hasmapto() 用于判断一个 {rhs} 是否有映射过。具体用法请用 :help 查问相应的函数说明。

- <expr> 这是通过一个表达式间接计算出 {rhs} 的用法。这是个相对高级的用法，将在下一节详细讨论。

*表达式映射

常规的映射定义 :map {lhs} {rhs} 只是简单的将一个键序列转换解析为另一个序列，所以这是一种静态的映射。如果在映射定义中结合表达式的思想，通过某种表达式计算出所要转换的 {rhs}，那就能极大地扩展映射的功能，达到静态映射所无法实现的灵活性。

有两方式在映射定义中使用表达式。一种是 <expr> 参数，另一种是表达式寄存器 @=。我们先讨论后一种方式。= 是一种特殊的寄存器，那么普通的寄存器又是什么概念呢？那就从宏开始说起吧。虽然乍看之下宏与映射的关系远着呢，但究其本质也是通过少量按键来实现需要大量按键的功能。

假设有这么个需求，将每两行连接为一行，怎么处理比较方便快捷。不妨打开在第一章示例生成的 ~/.vim/vimllearn/helloworld.txt 作为示例编辑文件吧，如果这个文件你未保存或丢失了，重新生成也是极快的。

vim 普通模式下有个命令 J 用于将光标当前行与下一行连接为一行，就是删去其中的回车符。如果光标初始在第一行，那么 J 就能将第一行与第二行合一行，光标停留在第一行；再按 j 下移到第二行，也就是最初的第三行，再按 J 合并……于是你可用这个按键序列 JjJjJjJj... 来将当前 buffer 内的每两行合并为一行。

这都是些重复按键呀，可以用宏来节省操作呢。假设撤销刚才讨论的操作，从最初打开的 helloworld.txt 重新开始，（普通模式下）请依次按这些键 qaJjq：

- q 是录制宏的命令，qa 表示将宏保存到寄存器 a中；
- Jj 就是刚才我们讨论的手动操作，将当前行与下一行合并，再将光标下移一行；
- q 再一个 q 表示结束录制宏。

现在我们已经有了 a 宏，就可以用 @a 命令播放这个宏了。可见其效果与在录制时的操作 Jj 是一样的。然后我们可以进一步在播放宏的命令之前加个重复数字。因为原来的 helloworld.txt 有 100 行，录制宏时合了两行，尝试播放宏时又合了两行，所以还需要再合并 48 次。用这个命令 48@a 就可以瞬间将剩余的文本行两两合并了。也可以使用 48@@ 命令，因为 @@ 是表示播放上一次播放过的宏。

（注：上述操作要产生相同结果，需要未打开折行选项，即 :set nowrap，或没有将 j 映射为 gj 或其他，同时 J 命令也未被映射）

那么宏到底又是什么，宏里面到底保存了什么神秘的东西。其实它一点都不神秘，宏就是一个寄存器而已。你可以用 :reg 命令（全名:registers）查看所有寄存器的内容，

或者特定地 :reg a 查看寄存器 a （宏 a ）的内容。可见它就是保存着 Jj 这两个字符而已。可以将它粘贴出来再确认下 o<Esc>"ap：

- o<Esc> 表示用 o 命令打开新一行，然后用 <Esc> 回到普通模式。如果你按刚才的批量宏操作后，光标应该位于 buffer 的最后一行；此时在最后新加了一空行，光标也在这空行上。
- "ap 粘贴命令 p 应属常见，在这之前先按 "a 表示从寄存器 a 中粘贴内容。

执行完这个命令后，就会发现已经将寄存器 a 的内容 Jj 粘贴到当前 buffer 末尾了。 常规寄存器有 26 个，即以 a-z 字母命名。我们可以试试其他寄存器，比如先用 v 选定 Jj 这两个字符，再用命令 "by 将这两个字符复制进寄存器 b 中。你可以用 :reg 命令再次查看下寄存器内容，确认 a 与 b 两个寄器都保存着 Jj 了。

题外话：我们平时使用复制命令 y 与粘贴命令 p 都不会加寄存器前缀的，这时它们使用的是默认寄存器，其名就是双引号 "，它其实是关联着最近使用的寄存器，与最近使用那个寄存器内容相同。可以在当前行继续尝试 p 命令与 ""p 命令（或在使用每个命令之前先输入一个空格，分隔内容方便查看），可见它们都粘贴出了 Jj。此外还有大写字母的寄存器，但它们不是额外的寄存器，只是表示往相应的寄存器中附加内容。比如若 v 选定 Jj 内容后，再按 "Ap ，就表示将这两字符附加到原来的 a 寄存之后了。可以用 :reg 查看 a 寄存器的内容已变成 JjJj 了。

为了说明宏即是寄存器，先用 q! 强制关闭当前的 helloworld.txt 而不保存，再重新打开原始的有 100 行的 helloeworld.txt。如果光标不在首行（vim 有可能会记住 光标位置的）则用 gg 回到首行。然后直接用命令 50@b，看看会发生啥。没错，这命令也将 buffer 内的文本行两两合并了，相当于执行了 50 次 Jj 命令。

所以 @a 或 @b 操作，正式地讲不叫“播放”宏，而是“读取寄存器，将其内容当作普通命令来执行”。其实，当作普通命令来执行的内容，不仅可以放在内部寄存器，也可以放在外部文件中。比如，只将 Jj 这两个字符保存到一个 Jj.txt 文件中，然后执行 ex 命令 :source! Jj.txt。当 :source 命令之后加个 ! 符号，就是表示所读的文件不是当作 ex 命令的脚本了，而是当作普通命令的“宏”了。在这个命令之前，请将光标移到首行，至少不要末行，否则就看不到 j 的效果了。同时由于这个文件只保存了一组 Jj，所以它只合并了两行。不过普通命令的序列组合可读性比较差，且很大程度地依赖操作上下文，所以一般不会保存到外部文件，临时录制保存到寄存器较为常见。当然你也可以先简单思考一下如何组织操作序列，明确地写出来，再复制或剪切到某个寄存器中。

当明白了 @a 的执行意义，也就能更好地理解 @= 的意义了。这里，= 与 a 一样是个寄存器，这个特殊寄存叫做表达式寄存器。

请在普通模式下，按下这两个键 @=，此时光标将跳到命令行的位置，不过前面不是 :， 而是 = 了。vim 在等待你输入一个有效的表达式，再按回车执行。比如输入 "Jj"<CR>，这里 <CR> 表示回车结束输入并执行，注意 "Jj" 需要引号括起，这样它才是个字符串常量表达式，否则若裸用 Jj，回车后 vim 会报错说 Jj 是个未定义 变量。

然后这整个按键序列 @="Jj"<CR> 的效果是什么？就是与普通命令 Jj 一样，合并两行并下移。可以用 :reg 查看寄存器 = 中的内容也正是 Jj。所以，@= 的意图是让用户临时输入一个表达式，vim 将计算该表达式的值，然后将结果值（应是字符串）

当作普通命令来执行。如果 @= 之后直接回车，不输入表达式，则延用原来保存在 = 寄存器中的值。

当你终于明白了 @= 的意义之后，就可以用 @= 来构建表达式映射了（终于回到正题了）。例如：

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

这样就可以用快捷键 \j 来“合并两行并下移”了。当然了，在这个简单的特定实例中，所谓快捷键 \j 其实并不比直接输入 Jj 快多少。那个映射命令似乎也可以直接写成 :nnoremap \j Jj。然而问题的关键是，在 @= 与 <CR> 之间，可以使用几乎任意合法的 VimL 表达式（即使不是所有），而不会是像 "Jj" 这样无趣的常量表达式。

举个实用的例子：

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

这个映射是说用空格键来切换折叠，即相当于命令 za，但如果当前行根本就没有折叠，那就无所谓切换折叠了，那就换用命令 } 跳到下一个空行。这里用到了条件表达式 ?:，我在脚本中很少用这个，不必省 if else 的输入，但在定义一些映射时条件表达式却是极简捷实用的。

在插入模式下（包括命令行模式），不是用 @ 键调取寄存器，而是用另一个快捷键 <C-R>。比如 <C-R>a 就表示将寄存器 a 的内容插入到当前光标位置上。如果用 <C-R>= 就表示将要读取表达式寄存器的内容了，此时光标也会跳到命令行处，允许你输入一个表达式后按回车，vim 就将表达式的计算值插入到光标处。例如：

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

它定义了一个映射，使用快捷键 <F2> 在当前光标处插入当前日期（请参阅 strftime() 函数的用法）。

然后再来看 <expr> 参数的意义与用法，比如以下两个映射定义是等效的：

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

可见，在使用了 <expr> 参数后，@=<CR> 就没必要了，直接将后面的 {rhs} 参数部分当作一个表达式，vim 首先计算这个表达，然后将其结果值当成真正的 {rhs} 参 数来解析为按键序列。

再尝试将上面那个空格切换折叠的快捷键改写成 <expr>

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

（注：我在 vim8.0 中测试该映射有效，但在 vim7.4 中同样的映射无效，可能在低版本中 <expr> 对条件表达式的 ?: 的支持不完全，但对于其他简单表达式无问题）。

除了应用条件表达式，当计算 {rhs} 需要涉及更复杂的逻辑时，还可以包装在一个函数中，那就几乎有着无限的可能了。仍以切换折叠的示例，改写成函数就如：

```
: function! ToggleFold()
:     if foldlevel(line('.')) > 0
:         return "za"
```

```
:     else
:         return "}"
:     endif
: endfunction
:nnoremap <expr> <Space> ToggleFold()
```

不过要注意，VimL 函数的默认返回值是数字 0，如果在函数中忘了返回值，或在某个分支中忘了返回值，那就可能导致奇怪的结果。例如，将上面的 ToggleFold() 函数改 写成：

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

假装忘了返回 l:rhs，那么快捷键 <Space> 将取得 ToggleFold() 的默认返回值 0，就是移到行首的意思了。取消 :return l:rhs 行的注释，可使之恢复正常使用。

当然了，用于表达式映射 <expr> 的函数还是有些限制的： * 不能改变 buffer 内容 * 不能跳到其他窗口或编辑另一个 buffer * 不能再使用 :normal 命令 * 虽然可在函数内移动光标，以便实现某些逻辑，但在返回 {rhs} 后会自动恢复光标 ，所以移动光标是无效的。

总之，映射的表达式函数尽量保持逻辑简明，以返回一个字符串作为 {rhs} 为主，避 免在其内执行有其他副作用的操作。更多内容请参考帮助 :help :map-<expr> 。

*命令后缀映射

定义命令后缀映射的命令是 :omap，当然最好用 :onoremap。要能定义有趣的命令后缀映射，首先就要理解命令后缀模式（Operator-pending，直译操作符悬挂模式）。

Vim 普通模式下的许多命令都是“操作符+文本对象”范式。比如最常见的 y d c 就 是操作符，当你按下这几个键之一后，就进入了所谓的“命令后缀”模式，vim 会等待你输 入后续的操作目标即文本对象。文本对象包括以下两大类：

1. 使用移动命令后光标扫描过的文本区域，即光标停靠点与原来光标位置之间的区域。
2. 预定义的文本对象，常用的有：

- ap ip 一个段落，段落由空行分隔，ap 包括下一个空行，ip 不包括。
- a(i(或 a) i) 一个小括号，a- 表示包括括号本身，i- 只是括号内部部分。
- a[a] a{ a}，i[i] i{ i} 与小括号类似。
- a" a' ，i" i' 与小括号类似，但是由引号括起的部分。

Vim 允许用户分别独立定义操作符与文本对象，然后任意组合。命令后缀映射就是可用 :omap 自定义文本对象。

还是举个例子。假如你需要经常操作双引号的字符串，觉得每次用 i" 略麻烦，因为它实际上是三个键，还要按个 Shift 键呢。你想选个单键来代替这三个键，比如说 q 键吧。首先，你可能尝试作如下映射定义：

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

然而，这只是个普通模式下的映射，并非命令后缀模式下映射，它不具备普适性。这里只定义了 dq 与 cq 就表明只能用这两个快捷键，但 yq 就无效了（复制字符串？），其他自定义的操作符当然也就无效。

然后试试改成一个命令后缀映射：

```
:onoremap q i"
```

这样，cq dq 与 yq 都有效了，如果你知道如何自定义操作符，它对自定义操作符也有效。

一个功能更丰富的例子请参考我写的一个小插件： https://github.com/lymslive/autoplug/tree/master/autoloa
在命令后缀模式下，单键 q 不仅可以模拟 i" 与 a"，还可以模拟 i(与 a(等括号对象（基于一定的上下文与优先级判断）。它的映射命令如下：

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

不过它所调用的函数实现略复杂，不便全部引用，有兴趣的请参阅源代码。

总结下命令后缀映射的机制，对于 :onoremap {lhs} {rhs} 映射。首先将 {rhs} 当 作普通模式下命令（按键序列）执行。如果执行后 vim 仍在普通模式下，且移动了光标，则将前后两个时刻的光标位置之间的区域当作文本对象。如果执行后在可视模式，则将选择部分的文本当作文本对象。内置命令 dw dp 类似前一种情况，而 da(di(类似后一种情况。

命令后缀映射的另一方面是操作符映射。也可以称之为命令前缀映射吧。这样，很多普通模式下的操作就可理解为“命令前缀”与“命令后缀”的组合了。定义满足这样特性的操作符的映射要分两步：

1. 设定选项 operatorfunc，其值一般是个函数名，用该函数来执行相应的工作。
2. 用命令 g@ 激活这个函数调用。

当然了，不要将这两步分开，如果单独将 operatorfunc 选项设置放在 vimrc，那 就只能定义一个操作符了。最好是类似如下定义：

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

就是临时设定 operatorfunc 的选项值，然后激活它。这样就能为不同的 {lhs} 定 义为不同操作符了。

操作符函数 OperaFunc() 有一定的规范。它收受的第一个参数表示文本对象的选择模式（即三种可视模式之一），这个参数是该操作符后面所接的文本对象自动传递给它的，其值为以下三种，在函数内可根据不同值作不同处理： * “line” 行选择模式 * “char” 字符选择模式 * “block” 列块选择模式

同时，在该函数内可利用 '[与 '] 这两个光标标记（mark）取得所操作文本对象的范围。即相当于文本对象的选择范围，加上参数一所指示的选择模式，就获得了足够的信息来操作文本对象了。

缩写映射

缩写也是一种映射，不过只用于可输入模式下。包括插入模式与命令行模式，以及不太常用的替换模式。其命令与映射也类似，不过将 map 换成 abbreviate，如：

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

也包括定义（退化参数则列表查询）、删除一个、清除所有缩写的命令。同样可以用 nore 限定，与模式前缀限制（但只有 i 与 c分别表示插入模式与命令行模式）。

缩写的含义是当你输入 {lhs} 时，自动替换为 {rhs}。不过由于在插入模式，字符是连续输入的，所以还有一些限定规则才能让 vim 识别刚才输入的几个字符是某个缩写的 {lhs}。

Vim 支持三类缩写，根据 {lhs} 中关键字位置区分。所谓关键字就是 iskeyword 选 项，一般认为数字、字符是关键字，其他标点符号与空白不是关键字。

- 全关键字（full-id），即 {lhs} 全部由关键字组成。必须完全匹配，即 {lhs} 之前不能有其他关键字。
- 关键字后缀（end-id），最后一个字符是关键字，前面的都不是关键字。
- 非关键字后缀（non-id），最后一个字符不是关键字，前面的可以是任意字符（空格与制表符除外）。

其中，全关键字是最常用的缩写，最直接的想法是用它来纠正拼写错误，如：

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

下面两例是另外两类缩写：

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

在使用缩写时，还要输入一个额外的键来触发识别缩写，这也叫缩写的展开。一般地，输入一个非关键字后，就会试图向前回溯寻找是否有缩写。最常用的是格式与制表符，还有离开插入模式的 <Esc> 与离开命令行模式的 <CR>。当缩写展开后，这个触发字符也同时会插入在被展开的 {rhs} 后，如果这不是想用的效果，可用一个快捷键 <C-]> 作为纯粹的缩写展开，而不会插入额外字符。

缩写同样支持 <buffer> 与 <expr> 参数。例如：

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

这两个缩写定义是等效的，在你输入 “today=” 之后（再空格或<C-]>等触发）就会替换为今天的日期。

那么它与插入模式下的映射又有什么不同呢：

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

如果把 “today=” 定义为映射的话，那么在输入前面几个字符 “today” 之前都不会上屏，接着输入 “=” 后立即上屏。这个体验并不好，因为你即使输入 “to” 时，vim 也会等 待，根据后续字符才能决定是否当作映射处理。

而定义为缩写的话，展开之前的字符是直接上屏的，是否展开的决定延迟，且可由用户决定是否展开。如果用户想抑止 “today=” 的展开，比如确实想在这个字符串之后输入个空格，则可用 <C-v><Space> 输入下一个空格。<C-v> 是插入模式下的转义快捷键，它后面接入的按键都屏蔽了其特殊意义，就按其字面字符输入。

结语

使用映射，除了一些基本的命令语法技巧外，更重要的是自己的统一习惯。可以多多凝视一下你的键盘布局，想想定义哪些快捷键自己会觉得比较方便与舒服。合适的快捷键对于每个人可能会有不同，不过有些键强烈建议不要重映射，请保留其默认意义：

- 数字不要被映射，数字用于表示命令的重复次数。
- 冒号 : 进入命令行不要改，当然如果觉得冒号不好按，可以将其他键也映射为冒号。两样建议保留的键是 <Esc> @ 键。
- 插入模式下的 <C-v> 与 <C-r>。Vim 的插入模式的默认快捷键确实不如普通模式方便，于是有些用户想把 Emacs 那套快捷键映射过来。或者 Window 用户想将 <C-v> 当作粘贴使用。然后这两个键在 Vim 映射中确实有特殊意义，经常能用来救急，还是保留的好。此外 <C-o> 是临时回到普通模式使用一个普通命令，也是很有用的，尽可能保留。

另外，关于 <Leader> 的使用。如果基本只用一种映射前缀，使用 <Leader> 是方便 的。但如果使用了多个 <Leader> 以对应不同类别的快捷键，则不太建议使用 <Leader>，直接写出映射前缀字符就是。毕竟 mapleader 是个全局变量，若要经常改变其值，就不容易维护了。

除了映射与缩写，Vim 的自定义命令与自定义菜单的用法与思想也是类似的。自定义菜单是只用于 gVim 的，本教程不打算介绍，而自定义命令将在一下节介绍。

第三章 Vim 常用命令

3.3 自定义命令

命令语法

定义命令与定义映射的用法其实很相似：

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

只不过在使用自定义命令时，{lhs} 是直接输入到命令行中的，当你按下回车时，vim 就将 {lhs} 替换为 {rhs} 再执行。所以这在形式上与下面这个映射等效：

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

当然，由于 :command 所支持的参数与 :map 大相径庭，并不期望你真的按这方式将自定义命令改成映射。实际上，Vim 的帮助文档中这样描述自定义命令的语法的：

:command {cmd} {rep}

:command! 加个叹号修饰则表示重新定义命令 {cmd}，否则若之前已定义 {cmd} 命令，:command 原版会报错。这是为了保护已定义不被覆盖，当你确实要覆盖时，请加 ! 后缀。在实践中，一般都是在脚本中定义命令，建议只用 ! 即可，尤其是在开发阶段需要调试脚本时，加上 ! 方便很多。

大部分命令的 ! 修饰版都是表示强制执行，忽略错误的意思。但上一节介绍的 :map! 的意义太奇葩，建议直接忘记 :map! 的用法。

:command 命令的退化用法是一致的： * :command {cmd} 列出以 {cmd} 开头的自定义命令； * :command 列出所有自定义命令；

Vim 的内置命令都是小写的（除了 :Next 与 :X :Print），所以要求自定义命令名 {cmd} 只能以大写字母开头，其后就类似 VimL 变量名的要求了。然而也不建议在命令名中使用数字，因为这可能与数字参数混淆。

内置命令可以缩写（这与上节的缩写映射不是同个东西），在没有歧义时，只要输入命令名的前几个字母就可以了。自定义命令 {cmd} 同样可获得此基本福利。不过内置命令还有更好的福利，就是钦定的缩写，比如 s 是替换命令 substitute 的缩写，但它不会与 set 发生歧义，而 set 的缩写是 se。自定义命令却无此特性，只能按基本规则，输入尽可能多的前缀字符来达到唯一确定命令名的目的。不过缩写只建议在命令行中使用，在脚本中尽量使用全名。

命令属性

在自定义命令时，可支持多种属性，就像 :map 的特殊参数（用 <> 括起来的）。但是在 :command 中，以一个 - 引导一个属性（更像 shell 命令行的选项）。所有属性必须出现在命令名 {cmd} 之前。

- -buffer 局部命令，只能用于当前 buffer。
- -bang 该自定义命令允许有 ! 后缀修饰。
- -register 第一个参数允许是寄存器名。
- -bar 该自定义命令后面允许用 | 分隔，接续另一个命令。在这种情况下，{rep} 参数内就不能有 | 了，否则会出现解析歧义。

以上这几个属性，只有是 -buffer 是常用的，并且建议能局部化时尽量局部化。其他的属性则较少用到。-bang 与 -register 只相当于某种特殊参数，而在同一行中用 | 使用多个语句（命令）的骚操作，能不用尽量不用。

然后，命令还支持几个复杂的属性，用 -attribute=value 表示，允许为属性指定值，要注意的是等号前后没有空格，而将整体当作 :command 命令的一个参数。

- 参数个数，自定义命令 {cmd} 允许多少个参数：
- -nargs=0 这是默认行为，不指定该属性就表示命令不接受参数；
- -nargs=1 仅接受一个参数；

- -nargs=* 接受 0 或多个参数；
- -nargs=? 接受 0 或 1 个参数；
- -nargs=+ 接受 1 或多个参数。

按常规用法，多个参数用空格分隔（或制表符）。但如果只有一个参数，末尾的空格会被认为是参数的一部分。否则若要参数中包含空格，请用 \ 转义。

- 范围数字释义，是否允许在命令之前加上一个或两个（以逗号分隔）数字：
- -range 允许两个地址参数或一个数字参数。不加该属性时，自定义命令默认不接收数字或地址参数。但这只是允许，可选加或不加，也不提供默认数字或地址。
- -range=% 允许地址参数，且默认是全 buffer，相当于 1,$。
- -range=N 允许一个数字参数，默认是 N，只能用在命令名之前。
- -count=N 与 -range=N 类似，不过数字参数不仅可以出现在命令名之前，也可以出现在命令名之后（相当于第一个参数）。-count 与 -count=0 等效。不过 注意，-range 属性与 -count 属性是互斥的，最好只用其中一个属性。
- 特殊地址. $ % 所表示的范围（在允许 -range 时）：
- -addr=lines 这也是默认行为，取当前 buffer 文本行的范围。
- -addr=arguments 指打开 vim 时命令行的文件名参数（其实也可以更改）。
- -addr=buffers 指所有打开过的 buffer。
- -addr=loaded_buffers 仅指当前加载的 buffer，在某个窗口中显示的 buffer。
- -addr=windows 取所有窗口列表的范围，仅限当前标签页。
- -addr=tabs 取所有标签页范围。

注意，-addr 属性必须要与 -range 联用才有意义。它要说明的是当命令的地址参数使用 .（当前）$（最后）%（所有）是参照什么集合而言的。例如定义如下命令：

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

则使用命令时，:.,$CmdA 表示用命令 CmdA 处理当前 buffer 内当前行到最后一行之间的文本行。:CmdB 表示处理所有 buffer，因为 -range 的默认范围是 % 表示 所有，而 -addr 表示所有的集合是指所有 buffer。同样，:.,$CmdT 表示处理从当 前标签页到最后一个标签页，虽然 -range=% 表示默认所有，但使用时可以自己加个特定的地址参数呀。

命令补全

自定义命令还有个最复杂的属性，是有关补全特性的。值得单独拿出来讨论。

Vimer 初学者倾向于使用映射，可能较少用到自定义命令。但是随着对 Vim 深入使用与 理解，可能就会发觉键盘的映射资源是有限的，尤其是要有规律地组织许多容易记住的映

射会有瓶颈。这时不妨将眼光投入到自定义命令中。虽然使用命令没有映射那么快，但只不过多加冒号与回车，就几乎有了几限的扩展可能。而且，在命令行中，不仅命令名可以补全，命令参数也可以补全，这就大大减少了记忆负担。

-complete 属性就是用于指定命令如何补全参数的，其取值范围非常广，这里仅介绍几种主要的补全行为，全部列表请参考 :help :command-complete:

- -complete=file 按文件（包含目录）补全，就像 :edit 命令按 <Tab> 后会补 全文件名那样。
- -complete=option 补全选项名。
- -complete=help 补全帮助主题。
- -complete=shellcmd 补全外部 shell 可用的命令。
- -complete=tag 补全标签，类似 :tag 所需的参数。
- -complete=filetype 补全文件类型名。

总之，如果自定义命令期望它的参数是某一类意义上的参数，就可以指定 -complete 属性为相应的值，以方便输入参数。当然，如果你定义的某个命令要实现比较复杂的功能，vim 预设提供的补全行为都不满足要求的话，还可以指定一个函数来实现补全。

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

这也叫做自定义补全。要注意的是，= 与 , 前后都没有空格，在 custom, 或 customlist,后直接接一个函数名。

当 -complete 属性值是 custom 时，函数要求返回一个以回车 \n 分隔的字符串，每一行是一个候选补全项。且 vim 会自动匹配比较光标前已经输入的部分参数前缀，进行一些过滤。

当 -complte 属性值是 customlist 时，函数要求返回一个列表，每个元素是候选补全项。但 Vim 不会自动对参数前缀过滤，可能要求用户自己在函数中过滤。

在这两种情况，补全函数的定义都是类似的，它应该接收三个参数： 1. a:ArgLead 光标之前的部分参数前缀， 2. a:CmdLine 整个命令行文本， 3. a:CursorPos 当前光标在命令行的位置（按字节计，从1开始）。

当用户按下补全键（一般是<Tab>），Vim 会自动将这三个参数传给自定义补全函数。用户在这个函数实现可利用这三个参数所提供的信息（也许不一定要用到全部），返回合适的候选补全项。

命令实现

我们将自定义名之后的 {rep} 参数部分称为命令实现。它可以是一串简单的替换文本，但真正有趣的是它可用一些特殊标记来表示特殊的或动态的内容。这里的特殊标记也用尖括号 <> 括起，所支持的有意义的标记可能依赖于前面的的命令属性。

- <line1> <line2> 分别表示地址参数的两个数字（一般是第一行与最后一行）。含 -range 属性的命令才能接收这两个参数。
- <count> 就是由 -count 属性提供的数字参数。
- <bang> 支持 -bang 属性的命令，如果使用时加了 ! 修饰，则在 {rep} 中的 <bang> 标记转换为 ! 字符，否则就没任何效果。

- <register> 或简写为 <reg>，支持 -register 属性的命令，表示可选的寄存器参数；否则也没任何效果（加上引号 "<reg>" 才表示空字符串）。
- <lt> 代表左尖括号 <，避免尖括号的特殊意义。比如想在 {rep} 中字面地呈现 <bang> 这几个字符串，而不是转化为 ! 字符，就可用 <lt>bang>。

先举个简单的例子，我们已经知道 :map! 命令是列出某类映射。虽然上文说过应该忘记这个命令，不过正因为它安全无害，不妨再拿来作为演示讲解。首先定义这个命令：

```
: command! MAP map
```

这个自定义命令似乎很无趣，不过用大写版的 :MAP 代替内置的 :map。请试试在命令中输入 :MAP 并回车执行，其结果与直接使用 :map 是一样的。试试 :MAP! 呢？Vim 会报错，说这个命令不支持 !。那么重定义一下这个命令：

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

现在，应该 :MAP 与 :MAP! 命令都可以使用了，并且分别与 :map 与 :map! 等价。这就是 <bang> 用于命令实现参数 {rep} 中的代表意义。同时，如果你没有定义其他以 MA 开头的命令，那么我们这个自定义命令简写成 :MA 或 :MA! 也是可以的。

由于这个自定义没有加 -nargs 属性，默认是不能接收参数的，所以若试图用 :MAP lhs rhs 来定义映射会失败。但是，加了参数属性后，又如何在 {rep} 中使用相应的 参数呢？这就是 <args> 标记的用途，同时这有多个变种：

- <args> 将用户在自定义命令后输入的参数原样替换到 {rep} 中。不过若命令还有 -count 或 -register 属性的话，前面的属性应该由 <count> 或 <reg> 捕获 ，而 <args> 只表示剩余的参数。
- <q-args> 与 <args> 一样，先捕获所有参数，然后将所有参数用引号括起来作为一个字符串表达式参数。如果没有参数，这将是一个空字符串（包含引号如 ""）。
- <f-args> 也与 <q-args> 一样，只不过将捕获的参数分隔成适用于函数调用时小括号内的参数列表，所以是将每个参数分别引起，并用逗号分隔。这在 {rep} 实现中 调用一个函数中非常有的。如果没有参数，则所调用函数的小括号内也没有任何东西，即以空参数调用。

现在继续来改造我们的自定义命令 MAP：

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

这样，:MAP 与 :MAP! 可以继续用，而且也可以用它来定义映射了，例如：

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

这里，用自己的 :MAP 来定义一个映射，将 x 删除一个字符的功能改为删一行。不过由于只为试验，所以加 <buffer> 定义成局部映射（注意区别，定义局部命令用 -buffer 语法）。

由于我们在定义 MAP 时允许它接收任意个参数 -nargs=*。所以在 :MAP <buffer> x dd 这个使用场合下，:MAP 的所有参数 <buffer> x dd 替换在定义 MAP 时 <args> 的位置上，也就相当于执行 :map <buffer> x dd。可以试下执行完，再按

x 是不是实现了预期效果，同时也可以用 :MAP x 或 :map x 查看下将 x 定义成啥样的映射了。

在这个示例中，如果将定义 MAP 时的 <args> 改成 <q-args> 或 <f-args> 的话，结果就不正确了，不能仿拟 :map 命令了。在实现复杂命令时，后两个参数变种标记才更有用，作为函数调用的参数。不过这较为复杂，留待下一小再论。这里先探讨一下 <register> 参数的使用，假设继续为 MAP 命令添加这个属性：

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

先将原来定义的 x 映射删除：:unmap <buffer> x。然后再用新的 :MAP 命令定义 x 映射，不过在参数 <buffer> 前额外加个参数 n：

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

结果是相当于只定义了普通模式下的映射 :nmap <buffer> x dd。你可以用 :map x 查看一下 x 的映射定义确认。并且对比一下 :MAP <buffer> X dd 不加 n 的用法 。

结论就是 <register> 不过是捕获了第一个参数，<args> 捕获其他参数。而 MAP 的定义 <register>map<bang> <args> 表明是将第一个参数直接拼在 map 之前作为 映射命令的模式前缀限定，而将其他参数用空格分开后作为 :map 命令的参数了。

这样看来，<register> 似乎很名符实呀。那么我们再尝试下将 un 作为 :MAP 的第一个参数，看它会不会变成 :unmap 用于删除映射：

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

然而，这次 vim 报错了，提示 umap n <buffer> x 不是一个命令。由些可见，<register> 只捕获的第一个字母 u，然后将剩余的东西都当成 <args> 了。因为寄存器名都是一个字母啊。

vim 有些内置命令如 :del :yank :put 支持后面接一个寄存器名（比如 a），表示对相应的寄存器操作，相当于普通模式的命令 "ad "ay "ap。自定义命令就可用 <regsiter> 实现类似的特性，使得自定义命令能像内置命令一样使用。只不过，<register> 只能捕获参数中的第一个字母，把它当成是寄成器名，传给 {rep} 实现 部分，却无法控制 {rep} 如何处理这个字母。因为 :map 命令的模式前缀限定恰好也只是一个字母，所以我们的 :MAP 就可以用 <register> 进行伪装了。你可以自行尝试 :MAP i :MAP c 等用法应该也是有效的。

上一节也提前，使用映射命令，尽量使用更安全的 :noremap，所以再重定义命令：

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

要测试这个命令是否有效，可定义如下映射：

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

再按 x 看看是否能正确只删除两个字符，还是会发生无尽循环故障（如果有这问题，按 <Ctrl-c> 中断即可）。

再次提醒：这里讨论不断“优化” :MAP 命令，只为说明 :command 自定义命令的用法与机制。正常使用 vim 下，应该没必要定义这么个命令呀。

自定义命令调用函数

除了很简单的命令，可以调用 vim 既有的内置命令（可能进行必要的包装修饰）外，大多实用的自定义命令，都是通过调用函数来实现命令要求的功能。这并仅可以实现很复杂的功能，也容易扩展，还使得用法简明易记，因为它一般如下的形式结构之一：

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

当使用自定义命令 {cmd} 时，它后面的命令行参数就会传入实际工作的函数 WorkFunc() 中。<f-args> 按空格分隔多个参数，然后分别引为字符串参数传入，如果要在参数中包含空格，要用 \ 转义，要传入 \ 就要用两个反斜杠即 \\。而 <q-args> 则简单粗暴，将 {cmd} 的所有参数，也就是其后跟着的所有内容当空一个字符串参数传入。在 {cmd} 之后没有任何参数时，<q-args> 也至少传入一个空字符串参数（WorkFunc("")），但 <f-args> 就不传入任何参数了（WorkFunc()）。

注意：传入 WorkFunc() 的参数必定是字符串类型，但由于 VimL 弱类型与自动转换，如果一个参数像数字，那么在函数体内将它当作数字处理也完全没有问题。

按 <f-args> 方式调用函数更为常见。<q-args> 可能只用于比较特殊的需要，然后要自己在函数体内解析字符串参数。另外，<f-args> 只适用于函数调用参数，用在其他地方的意义不明显，且易出错。而 <q-args> 用于函数参数之外也可能是有意义的。本小节暂时不讨论 <q-args> 的使用。

使用 range

首先我们需要一个工作函数。不妨复用在 2.4 节讲述函数时使用的给文本行编号的示例函数吧，取那个支持 range 特性的版本，并改名为 NumberLine 重贴于下：

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

然后定义一个命令也叫 NumberLine，用以调用该函数，命令名与函数不需要相同，只是懒得另起名字，同时也想说明，命令与函数重名完全没问题，因为它们是完全不是同类概念：

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

注意到 NumberLine() 函数不支持显式参数，但可接收隐式的地址参数。而命令 :NumberLine 正好定义为支持 -range 属性，这就要将捕获的地址参数

<line1>,<line2> 放在 call 之前，由 call 把地址参数传给 NumberLine() 函 数的 a:firstline 与 :lastline。

现在我们就可以来试用这个自定义命令了。如果直接在命令行输入 :NumberLine 回车 执行，它会对当前 buffer 的所有文本行编号。因为 -buffer 属性的默认值 % 就表 示所有行，相当于 1,$。如果我们按行可视模式 V 选择几行，再按 :NumberLine，命令行中实际输入的是 :'<,'>NumberLine，它就只会对选择的行进行编号。

使用 count

接着讨论下与 -range 相似但互斥的 -count 属性。<count> 只有一个数字参数，即可放在命令之前，也可以放在命令之后（甚至对是否有空格分隔不敏感）。很多 vim 内置命令的数字表示重复次数，不过在自定义命令中，<count> 只负责捕获传递这个数字参数，并无法控制后续命令如何使用这个数字，就如 <register> 一样。

我们另外写个函数，用于对当前行及后面若干行进行相对编号，即当前行号是 0，下一 行是 1 等（类似 :set relativenumber）。

```
function! NumberRelate(count) abort
    let l:cursor = line('.')
    let l:eof = line('$')
    for l:count in range(0, a:count)
        let l:line  = l:cursor + l:count
        if l:line > l:eof
            break
        endif
        let l:sLine = getline(l:line)
        let l:sLine = l:count . ' ' . l:sLine
        call setline(l:line, l:sLine)
    endfor
endfunction

command! -count NumberRelate call NumberRelate(<count>)
```

同时也定义一个相应的命令。试试效果？如果直接运行 :NumberRelate ，由于 -count 的默认值是 0，所以只对当前行编号为 0。如果对选区运行 :'<,'>NumberRalate，给命令提供了两个地址 vim 只会将后面那个地址参数 '> 当作数字参数 <count> 传给函数 NumberRelate() 的参数。同时也可以手动输入数字如 :3NumberRelate 或 :NumberRelate3 都会对当前行及后面3行编号。其中 NumberRelate3 的写法可能会 有歧义，如果恰好还有个自定义命名叫叫 NumberRelate3。所以最好用 :NumberRelate 3 来调用。也正是这个原因，不建议在命令名中混入数字。

至于 Vim 为什么允许命令与数字参数粘在一起使用，主要是因为要快捷输入。很多最常 用的命令都是有单字母缩写的，而与数字参数的组合使用又极频繁。在这种情况情况下多 敲一个空格的性价比太低了（我的命令才一个字母呢），所以就把空格吃了吧。

这个示例也说明，自定义命令调用函数时，参数不一定要用 <f-args> 或 <q-args>，混入其他任何特殊标记也是可以的，只要展开替换后符号函数调用语法即可。再比如，call WorkFunc(<bang>) 是非法的，因为展开是 call WorkFunc(!)，但 call WorkFunc("<bang>") 是合法的，因为展开后是 call WorkFunc("!")。而

<count>（其实也包括 <line1> <line2>）可直接放入函数括号内，是因为它们会展开成一个数字。

使用 f-args

前面两例所用的函数都不接收参数，如果函数要求参数，就用 <f-args> 传入吧。假设更改为文本行编号的需求，在数字编号后还允许加个后缀字符，像 1. 1) 之类的， 同时可以定制分隔编号与原文本之间的空格数量。我们重写 NumberLine 函数，让它接 收两个参数：

```
function! NumberLine(postfix, count) abort range
    let l:sep = repeat(' ', a:count) "    count
    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>)
```

然后也重定义命令 :NumberLine，为其增加 -nargs 属性，然后用 <f-args> 传给 函数调用。注意虽然可以用 -nargs=1 限定允许一个参数，但不支持 -nargs=2 限定 恰好两个参数，只能用不定数量的 -nargs=* 或 -nargs=+。此时若只用 :NumberLine 命令执行，会报错说参数太少，加上两个命令行参数后如 :NumberLine) 4 就能正常工作了，这表示编号样式为 1) 然后接 4 个空格。

注意到 NumberLine() 函数虽然也有个 count 参数。但与上例不同，不能用 -count 属性与 <count> 参数。首先是因为 -count 与 -range 属性只能用一个，不能共存。其次这里的 count 参数与大多 vim 内置命令对数字参数的解释很有些不同，只是恰好用了这个形参名而已。因此不要滥用 <count> 参数，能直接用 <f-args> 是最简洁明了的。

如果工作函数 WorkFunc() 没有 range 属性，不处理地址范围的话，那么自定义命令时，也不要加 -range 属性，而后面的调用函数写法也更加简单。

另外，如果工作函数是脚本作用域的函数，如 s:WorkFunc()，则在 {rep} 部分中调 用写成 <SID>WorkFunc()，高版本的 vim 也可以直接用 s:WorkFunc()。不过上节的映射命令 :map，却只能用 <SID> 而不能用 s:。

*微命令实例

本节内容所用的命令示例，主要为阐述概念，也许并无实用性。我在大量使用映射后，也开始对命令有所偏爱了。为了使命令输入尽可能方便，我将常用命令也定义很短的几个大写字母，并称之为“微命令”。实现脚本放在了 github 上，有兴趣的可以参考，传送门在此：https://github.com/lymslive/autoplug/tree/master/autoload/microcmd

如果命令名较长，输入不便时，也可以继续使用映射来触发命令，甚至可以将最常用的命令参数也一并包含在映射中。

第三章 Vim 常用命令

3.4 execute 与 normal

为什么这两个命令值得单独拿出来讲，因为它们使得其他大部分 Vim 基本命令变得可编程，用 VimL 编程。不仅是更高层次上的流程控制，更可以控制单个命令的执行，控制所要执行的命令或参数。简单地说，就是可利用 VimL 语言的一切特性，拼接并生成将要执行的 ex 命令，然后真正执行它。

- `:execute` 将 VimL 的字符串（值）当作命令执行。
- `:normal` 用 ex 命令的方式执行普通命令。

基本释义：execute

还是通过例子来说明。`:execute 'map x y'` 相当于直接执行命令 `:map x y`。当然这 似乎没什么用，多套层 `:execute` 似乎写起来还更复杂。但是我们可以这样写：

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

似乎还更复杂了是不？然而，这背后的思想在于，`lhs` 与 `rhs` 都是变量，我们可以根据需求计算出它们值，然后再定义相应的映射。这就可以灵活地动态地执行 ex 命令了 。一般情况下，我们会把 `:execute` 命令写在脚本或函数中，比如写个叫 `s:BuildMap()` 的函数封装一下：

```
function s:BuildMap() abort
    let l:lhs = 'x'
    let l:rhs = 'y'
    let l:map = 'nnoremap'
    execute l:map . ' ' . l:lhs . ' ' . l:rhs
    "
    execute l:map l:lhs l:rhs
endfunction
```

`:execute {expr}` 这是 `:execute` 的正式语法，它后面接一个表达式。vim 首先计 算出这个表达式的值，一般期望它是个字符串，如果不是字符串也会自动转为字符串。然后执行这个字符串。

事实上它可以跟多个表达式，`:execute {expr1} {expr2}`，vim 会先求出各个表达式的值，再拼接成一个字符串，中间有个空格。如果你不确定这个自动拼接机制，或者不想在相邻表达式之间多加个空格，则可以用 VimL 的字符串连接操作符，一个点号 `.`，这样 就可以自己把握要或不要这个空格了。

在上个示例中，我们将函数内的局部变量直接赋值了（常量字符串），这仅为说明 `:execute` 的用法特征。更好的封装做法是利用函数参数，例如：

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

把函数体简化为一条语句了。当然更健壮的做法应该先检测一下 a:map 参数是否合法的映射命令，以避免一些灾难错误。而且真正的映射命令可能还不止由这三部分组成，还可能有很多类似 <buffer> 这样的特殊参数呢，不过这里暂不考虑了。

当把眼光向外拓展，函数参数怎么来？那就把 VimL 当作普通脚本语言（类似 python perl lua 这种脚本思想），根据需求计算变量的值，传递参数调用函数就可以了。

基本释义：normal

那么 :normal 命令又有何妙用。因为 VimL 本质上只是 ex 命令的组合，原则上在 vim 脚本中只能使用 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达成，但在 ex 命令行模式（或脚本中）却可能一时找不到对应的命令来实现相同功能，或者可以实现却写起来麻烦。

这时 :normal {commands} 命令就来帮忙了。它将其后的 {commands} 参数当成是在 普通模式下按下的字符（键）序列来解释。比如我们知道在普通模式下用 gg 跳到首行， 用 G 跳到末行。可有什么 ex 命令来完成这任务吗？有肯定有，至少可能调用函数 cursor() 来放置光标，但是用 :normal 似乎更简明：

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

要注意与 :execute 命令不同的是，:normal 的参数 {command} 它不是个表达式，它就表示字面上看到的字符。如果写成 :normal "gg" 反而错了，因为在普通模式下，前两个字符（按键）"g 是取寄存器 g 的意思呢。

:normal! {commands} 的叹号变种，表示后面的 {commands} 不受映射的影响。因为正常用户使用 vim 时都会在 vimrc 中定义相当多的映射，所以 :normal 命令会继续根据映射来再次查寻将要执行的（普通）命令。这往往使得结果不可预测，所以一般情况下建议使用 :normal! 而非 :normal。

不过，使用 :normal 还是有些限制的，毕竟不能完全像普通模式那样的使用效果。最重要的一点是 :normal 命令必须完整。如果命令不完整，vim 自动在最后添加 <Esc> 或 <Ctrl-c> 返回普通模式，以保持完整性。完整性不太好定义，那就举例说几个不完整的：

- 操作符在等待文本对象时不完整。如果执行 :normal! d 什么事都不会发生。因为在普通模式下 d 会等待用户继续输入文本对象。而用 :normal 来执行时，就无从等待，结果就是像按下 d 后又按下 <Esc> 取消了。但是 :normal! dd 能正确完 成删除一行的操作。
- 用 :normal 命令进入插入模式操作后，会自动 <Esc> 回到普通模式，不会停留在插入模式。例如 :normal! Ainsert something 会在当前行末增加一些字符串，但是整个命令结束后，不能期望它还在插入模式，它会回到普通模式。
- 在 :normal 后面用冒号进入命令行模式并输入一些命令，却不能以想当然的方式执行。比如输入 :normal! :map 后按回车，它并不会执行 :map 命令列出映射。因为它相当于在命令行输入 :map 后按 <Ctrl-c> 取消了，并不是按回车执行了。你必须用个技巧将回车符添加到 :map 之后才行，直接按回车是执行 :normal! 这条 命令的意思。这样输入：:normal! :map^M 再按回车就可以了，其中 ^M 表示回车 符，通过按 <C-v><CR>两个键才能输入。

总之，:normal 命令执行完毕后，会保证仍回到普通模式。也因此不能通过 Q 键进 入 Ex 。

execute + normal 联用

正如上面看到，:normal 命令后的参数（普通命令按键序列），只适于可打印字符，对于特殊字符，须用 <C-v> 转义后才能输入，这不太方便。但是可用 :execute 命令 再套一层，因为它接收的字符串表达式，当用双引号引起字符串时，特殊字符可用 \ 转 义。比如为解决上面那个难题 :normal! :map:

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

但是，execute + normal 的基友组合，远不止是为了输入特殊字符这么简单。:execute 还可以使 :normal 也用上变量。例如，我们可以用 5gg 来跳到第 5 行 ，用 :normal 命令也能跳到特定行:

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

然而，你无法直接动态地改变 5 或 10 这个数字，借且 :execute 就可以了:

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

再举个例子，在第 1.2 节，我们在普通模式下生成了一个满屏尽是 “Hello world!” 的 文章，回顾如下:

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

现在，我们用 VimL 语言编程的思路，利用 execute + normal 重新生成。既是编程，封装成函数才好:

```
function HelloWorld(row, col) abort
    normal G
    let l:word = 'Hello World!'
    for i in range(a:row)
        normal! o
        execute 'normal! ' . a:col . 'a' . l:word
    endfor
endfunction
```

函数接收两个参数，分别表示生成多少行，与每行多少个“Hello World!”。在函数体中，:normal! G 先将光标定位到当前 buffer 末尾，以便在末尾插入许多 “Hello World!” 。然后对每一行循环，每行循环中，先用 o 命令打开新行，再用 :execute 拼接重 复多次的 a 命令。

你可以用函数调用命令 :call HelloWorld(100,20) 来达到 1.2 节的效果，并且可调用行列数生成不同规模的“Hello World!”。

*用 execute 定义命令

在上一节中，我们推荐了一种定义命令的常用范式：call WorkFunc(<f-args>)。这里再介绍另一种定义命令的有趣范式：

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

形式上只是把 :call 命令换成了 :execute 命令。将自定义命令 {cmd} 的所有参数打包传给函数 ParseFunc()，期望它返回一个字符串，再用 :execute 执行它。

这另有什么妙用呢？一般情况下，用 :execute 可能只想到用它来执行常规的 ex 命令，但是也并不妨碍它用于执行 VimL 的特殊语法命令。例如，:let 命令只能一次创建 一个变量，下面这种“连等号”的语法是错误的：

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

但我们可以试着自定义一个 :LET 命令，让它允许这个语法：

```
" 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>)
```

这代码有点长，适合保存在脚本文件中再 :source。先解释下函数 ParseLet() 的意 思：它首先将输入参数按等号（两边允许空格）分隔成几部分；将最后部分当作是值，其余每部分当作一个变量，然后构造命令用 :let 为每个变量赋相同的值；最后将几个赋值语句用 | 连接并返回，| 是在同一行分隔多个语句的意思。

有了 ParseLet 函数后，再定义一个命令 :LET，现在就可以尝试下连续赋值了：

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

可见 x y z 三个变量都已经被赋值为 1 了。最后一个 :echo 语句是为了显 示 :LET 如何工作的，实质上它转化为 let x=1 | let y=1 | let z=1 多个赋值语句了。

那么，新定义的 :LET 能否正确处理变量的作用域呢，我们写个函数测试一下：

```
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

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

我们在函数中也定义了 x y z 这三个局部变量。结果表明，用 :LET 定义的局部变量与全局变量也互不冲突的，可放心使用。

不过，:execute 命令毕竟还是有所限制的。只适合用于定义一些简单的“宏命令”，并不能妄图重定义一些复杂的语法结构。而且，:execute 的效率也不高。

第三章 Vim 常用命令

3.5* 自动命令与事件

前面章节介绍了自定义快捷键（:map）与自定义命令（:command），这都是响应玩家的主动输入而快速做些有用的工作。这也算是对 Vim 的 UI 设计吧。谁说只有图形界面才算 UI 呢，况且在 gVim 中的自定义菜单，也确实与自定义命令或映射很相似呀。

本节要介绍的自动命令，却是让 Vim 在某些事件发生时自动做些工作，而不必再手动激活命令了。当然了，自动命令在生效前，也是需要定义的。

自动命令的定义语法

自动命令用 :autocmd 这个内置命令定义，它至少要求三个参数：

: autocmd {event} {pat} {cmd}

- {event} 就是 Vim 预设的可以监测到的事件，比如读写文件，切换窗口等。
- {pat} 这是模式条件的意思，一般指是否匹配当前文件。
- {cmd} 就是事件发生且满足条件时，要自动执行的命令。

在一个命令中可以有多个事件，事件名用逗号分开，且逗号前后不能有空格。模式也可能以逗号分隔为多个模式。因为{event} 与 {pat} 都相当于是 :autocmd 的单个参数，其内不能有空格。但最后部分 {cmd} 可以有空格。

一般情况下，{cmd} 就是合法的 ex 命令，将它拷贝到命令行也能手动执行那种。不过 {cmd} 中可能含有一些特殊标记 <> ，在执行前会替换成实际值，这才大大增加了自动命令的灵活性，而非只能执行静态命令。

在 vim 内部，相当于为每个事件 {event} 维护了一个列表，每当用 :autocmd 为该 事件定义了一个自动命令，就将这个命令加到列表中。然后每当事件发生，就遍历这个命令列表，如果它满足相应的 {pat} 条件，就会执行这个 {cmd} 命令。

因此，每发生一个事件，vim 都可能自动执行许多命令。就比如文件类型检测与语法高亮着色，就是通过自动命令实现的。当你安装一些复杂插件，可能会自动执行更多的命令。而我们自己用 :autocmd 定义的自动命令，只是添加在原来的命令列表之后，做些自定义的额外工作。

与此前的 :map 与 :command 一样，退化的 :autocmd 是查询功能：

- :autocmd {event} {pat} 列出与事件及模式相关的自动命令。
- :autocmd * {pat} 列出满足某个模式的所有事件的自动命令。
- :autocmd {event} 列出与某事件相关的所有自动命令，不论模式。
- :autocmd {event} * 与 :autocmd {event} 等效，* 就表示匹配所有。
- :autocmd 列出所有自动命令。

叹号修饰的 :autocmd! 命令用于删除自动命令，参数意义与退化命令一样：

- :autocmd! {event} {pat} 根据事件与模式删除自动命令。
- :autocmd! * {pat} 只根据模式条件删除自动命令。
- :autocmd! {event} 只根据事件删除命令。
- :autocmd! {event} * 只根据事件删除命令。
- :autocmd! 删除所有自动命令。

但是，叹号也可以修饰完整的非退化的 :autocmd ，就如定义自定义模式一样：

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

它表示先将满足事件 {event} 与模式 {pat} 的所有自动命令删除，然后添加自动命令 {cmd} 。因此这是覆盖式的定义自动命令，此后，在满足相应事件与模式时，就只会执行这一个自动命令了。依前文介绍，在定义命令与函数时建议用覆盖式的叹号修饰命令 :command! 与 :function!。但对于自动命令，还是慎重用覆盖式的 :autocmd! ，因为可能无法从本条语句判断会覆盖掉什么自动命令。

自动命令组

自动命令组 augroup 是组织管理自动命令的有效手段。为理解自动命令组是有必要的，先回顾上一小节所介绍的自动命令机制，在未利用命令组的情况下，会发生什么不良后果。

因为 :autocmd 定义自动命令时是将其添加到自动命令列表末尾的，所以如果在脚本如 vimrc 中定义了自动命令，随后又重新加载了该脚本，那自动命令列表中就会出现两项重复的自动命令了。对于某些“安全”的自动命令，重复执行不外是浪费效率而已，但有些自动命令在第二次执行却有可能引发错误呢。

其次，用 :autocmd! 删除自动命令时，它是删除所有自动命令。即使加了事件与模式两个限制条件，也无法避免影响扩大化，因为别的插件或 Vim 官方插件也可能为相同的事件与模式定义的一些有用的自动命令啊。

为了解决这个管理问题，引入了自动命令组的概念。自动命令组名字是用以标记一个自动命令组的符号，取名规则就按 VimL 变量名的规范吧（虽然帮助文档中说似乎可以用任意字符串作为组名，除了空白字符），不要用奇怪的字符，同时也是大小写敏感的。然后两个特殊的自动命令组名 END 与 end 是保留的，有着特殊意义。

在不发生理解歧义下，我们就用自动命令组名表示一个自动命令组吧，且在本节中，不妨用“组名”来作为自动命令组名的简写吧。

于是，在定义自动命令的 :autocmd 命令中，还支持一个可选的组名参数，它紧接命令之后，而在 {envent} 事件之前：

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

正因为组名是 :autocmd 的第一个参数，可有可无，当省略时，第一个参数就是事件名了。所以我们选取组名时，还要避免与事件名（这是 Vim 预设的范围集）重名，以避免歧义。

在定义自动命令时，如果指定了 [group] 组名参数，就表示将所定义的自动命令添加到这个自动命令组中。你可以认为每个组都为不同事件维护了不同的自动命令列表，同一事件在不同组内关联着各自不同的命令列表。

对于删除自动命令的 :autocmd! 变异命令，也同样支持在第一个参数中插入可选的组名。在指定组名后，就表示只删除该组内的自动命令（当然可再限定事件与模式）。

那么，在缺省组名参数时，:autocmd 与 :autocmd! 又怎样工作的呢。其实它是针对当前组添加或删除自动命令的。那么当前组又是什么东西呢？它是用 :augroup 命令选 定的：

```
: augroup {name}
```

在执行这个命令之后，{name} 就是当前组名了。当 {name} 组名此前尚不存在时，也会自动创建一个组，然后再选择这个组作为当前组。此后 :autocmd 或 :autocmd! 若不指定组名参数，就用 {name} 替代了。

那么，在第一次使用 :augroup 选定当前组名之前，当前组又是什么呢？那就是默认组（default group）了。默认组没有名字，你要把它想象为空字符串也行。或者形式地说，默认组名是 END 或 end，因为在以下命令表示选择默认组名：

```
: augroup END
```

因此，在脚本中定义自动命令的一般规范是这样的：

```
augroup SPECIFIC_GROUP
    autocmd!
    autocmd {event} {pat} {cmd}
augroup END
```

首先选定一个组，紧接着用 :autocmd! 删除该组内原来所有旧的自动命令，然后用 :autocmd 重新定义新的自动命令，可能有多条 :autocmd 自动命令，最后用 END 选回默认的（无名）组。这样，即使这个脚本重新加载，这个组内的自动命令也正是在这块脚本内所能看到的这些自动命令了。

当然了，你的组名不要别的组冲突。建议依据脚本文件名或插件名定义组名，且用大写字母，因为组名很重要，但其实又不必写很多次，故用大写字母表示合适。而且，尽量把自定义命令写在一块，不要分散。

这样，在组内定义的自动命令就有了局部特性，相当于局部自动命令，而在组外的（无名默认组）自动命令，就相当于全局自动命令。在编程的任何时刻，都尽量用局部的东西，

少用全局的东西。就自动命令而言，除了直接在命令行临时测试下什么自动命令，在脚本插件中，永远不在默认的无名“全局”组定义自动命令。

另外提一点，退化的查询命令 :autocmd 在缺省组名参数时，不是依据当前组，而是列出所有组内的自动命令。这与定义或删除自动命令时的缺省行为不同。这也好理解，因为只是查询，还是希望尽可能查出更多，而修改操作，却要尽可能缩小影响范围。

还有，组名只影响定义与删除自动命令的操作，但不影响事件触发自动命令。即不管定义在哪个组内，事件触发时，并且检测满足模式后，就能执行相应的自动命令。

使用事件

Vim 会监测大量事件，详细列表请查看文档 :help autocmd-events，这里只介绍几种常用的事件。事件名不分大小写，然而建议按文档中的名字使用事件。

- 读事件。有很多相似但略有细微差别的事件，BufNewFile 指创建新文件，BufRead 指读入文件。一般用这两个就可以了。若有更多控制需求，可用 BufReadPre 与 BufReadPost，这些事件一般会在 :edit 等命令时触发。若用 :read 命令，可 触发 FileReadPre 与 FileReadPost 事件。
- 写事件。:w 写入当前文件时触发 BufWrite 事件，部分写入（如 '<,'>w file ）则触发 FileWrite 事件。
- 窗口事件。新建窗口触发 WinNew，进入窗口触发 WinEnter，离开窗口前触发 WinLeave 事件。
- 标签页事件。类似窗口事件有 TabNew TabEnter TabLeave。
- 整个编辑器启动与离开事件：VimEnter VimLeave。
- 文件类型事件，当 &filetype 选项被设置时触发 FileType。

举些例子。为了方便，直接在命令行中定义自动事件了，只为简单测试。不过首先也创建一个组吧，比如：

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

在这里，先是创建并选定 TEST 为当前组，然后什么也没干又用 END 选回默认组。此后我们定义自动命令时都将显式地指这在 TEST 组上操作。你也可以先不用 :augroup END，保持当前组为 TEST，只为了想在之后的 :autocmd 缺省组名？但是在命令行操作中说不定会触发加载其他插件，这样就会改变当前组名了。所以为了原子操作的独立性，还是先选回默认组吧，也避免后来忘了执行 :augroup END。

然后定义一个自动命令：

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

这里显式指定在 TEST 组内定义自动命令，:autocmd 只能使用已存在的组，所以我们之前才要用 :augroup TEST 然后又 :augroup END 的“空操作”。BufNewFile 与 BufRead 经常同时用，这样不管是打开编辑已存在的文件，还是新建文件都能触发。在 {pat} 部分我们先简单用 * 表示匹配所有。最后的 {cmd} 部分仅是打印一条消息。

现在请试试打开另一个文件，或切换另一个 buffer，看看会不会打出“Hello World!”的 消息。如果消息被其他后续消息覆盖而看不到，请用 :message

打开消息区（可能还须 用 G 翻到最后）再看是否有这个记录。

再定义另一个自动命令，在打开 vim 脚本文件中显示不同的消息：

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

然后用 :e $MYVIMRC 打开你的启动配置文件，看看有什么欢迎消息？似乎仍是打印“Hello World!”，而不是“Hello vim!”？那么请用 :echo $MYVIMRC 查看下你的配置文件是哪个文件，一般应该是 ~/.vimrc 或 ~/.vim/vimrc，它并不是以 .vim 作为后缀的文件名呢。所以不能匹配 *.vim 这个模式。

那么手动打开一个确实以 .vim 为后缀的文件再试试看吧，或者新建一个 vim 文件 :e none.vim。不出意外的话，你应该会看到两条消息，“Hello World!”与“Hello vim!”都 打印了，因为它确实同时满足刚才定义的两个自动命令啊，所以两个都执行了。然后再试 :e none.VIM，新建一个文件以大写的 .VIM 为后缀名。这也不会触发“Hello vim!”，可见文件模式是区别大小写的，它未能匹配到 .VIM。关于模式的细节，下一小节再详叙。

为了避免消息太多，我们先把刚才两个自动命令删除了，再定义另外一个自动命令：

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

这里，<afile> 表示在触发自动命令时，所匹配的那个文件名（一般是当前文件名）。再试试打开文件，会打印什么欢迎消息？

切记：在用 autocmd! 删除命令时，要加上组名 TEST，否则可能会删去一些定义在默认组的自动命令。

写文件事件也一样定义自动命令：

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

然后随便编辑一个文件，用 :w 写入，是否能预期的“bye …”消息。很可能看不到的。因为 BufWrite 事件是在开始写的时刻触发，然后写完后 vim 一般会自动再打印另一条消息显示写入多少字节。消息被覆盖了！但用 :message 再翻到末尾应该就能看到了。那么我们把事件改为写之后试试：

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

再看看写文件时会提示什么消息。顺便说一下，BufWritePre 事件与 BufWrite 其实 是等效的。如果没有特殊需要，建议用 BufWrite 比较简便。

然后再举个切换窗口的自动事件：

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

这里 winnr() 函数将取得当前窗口编号。定义完这两个自动事件后，请将你的 vim 分 裂出多个窗口，在窗口间切换，以及关闭多余窗口，看看会有什么消息提示（用 :message G 确认消息）。由此你应该能得到结论，切换窗口时先触发 WinLeave 事件，再触发 WinEnter 事件。

其他事件就不一一举例了，请自行对感兴趣的事件进行测试。然后在实际写插件或脚本时，若想实现某个自动功能，先查阅文档，找个合适的事件，理解它的触发时机。如果 Vim 没有提供合适的事件，可能自动命令就无能为力了。不过幸运的是，Vim 已经提供了大量 的事件，应该能满足绝大部分需求了。或者，当你功夫足够深时，可以从近似的事件入手进而曲线救国。

再次提醒，如果是在脚本中定义自动命令，请按以下规范写：

```
" save in somefile.vim
augroup TEST
    autocmd!
    autocmd BufNewFile,BufRead * echomsg 'hello ' . expand('<afile>')
    autocmd BufWrite * echomsg 'bye ' . expand('<afile>')
    autocmd BufWritePost * echomsg 'goodbye ' . expand('<afile>')
augroup END
```

在 :augroup 块内不必再指定 TEST 组名了，虽然也可以在每个 :autocmd 命令重复加上这个组名，但是建议省略。因为万一以后因为某种原因要改组名，却忘记了同步修改里面的每个组名，那就麻烦了。

所以，把 :augroup 与 :augroup END 当作像 :function! 与 :endfunction 一 样的独立单元块吧。只不过里面的命令不是由显式的 :call 调用，而是 vim 根据事件 自动调用了。于是，很显然地，自动命令组名应像（全局）函数名一样，不要与其他组名冲突。

在实用的自动命令中，{cmd} 部分一般是调用一个工作函数，以简化 :autocmd 的语 法，而把复杂的逻辑实现放在函数中。特殊标记如 <afile> 表示匹配的文件名，在触发自动命令时才展开。但有个例外，<sfile> 表示的是定义该自动命令时所在脚本文件（假设你不是把自动命令放在函数中定义，一般应该是这样）。同时，在 {cmd} 部分 也可以用 <SID> 表示当前定义脚本范围的元素，比如 s:Function。

文件模式

定义自动命令时 :autocmd 的第二参数（可选组名除外），即 {pat} 是文件模式的意思。它不同于正则表达式，而像是操作系统的文件名通配符。即 * 表示任意字符，? 表示单个字符。详细符号意义请查看 :help file-pattern。这里只强调几点需要注意的地方：

- 逗号表示多个模式的或意义。如 *.c,*.h,*cpp 表示 c/c++ 文件。
- 如果模式中没有路径分隔符 /，则只匹配文件名。
- 如果模式中包含 / 则要匹配文件全路径名。如 /vim/src/*.c 只匹配位于 /vim/src/ 目录下的 c 文件，这可能是 Vim 源代码的工程文件。而 */src/*.c 则匹配任意目录下的子目录 src/ 内的 c 文件，可能表示任意一 c 语言工程内的源 文件。
- 一些命令如 :edit 会将其参数内的环境变量（如$MYVIMRC）与特殊寄存器（如 % 与 #）展开，则在将实际文件名展开后再匹配自动命令中的文件模式。

如果文件模式 {pat} 用一个特殊参数 <buffer> 代替，则表示定义了一个只局部于特定 buffer 的自动命令。这又有几个变种：

- <buffer> 所定义的自动命令影响当前 buffer，即只有在当前 buffer 才能触发。
- <buffer=N> 这里 N 是一个数字，表示只影响编号为 N 的 buffer。用 :ls 命令或 bufnr() 函数可以查看 buffer 的编号，那算是唯一不变的 id。
- <buffer=abuf> 这里的 <abuf> 是在触发自动命令时的特殊标记，如同 <afile> 表示触发的文件，而 <abuf> 表示触发的 buffer 编号。这个参数只在当自动命令中定义另一个自动命令时有用。

例如，:autocmd BufNewFile * autocmd CursorHold <buffer=abuf> echo 'hold' 表示每当新建一个文件（BufNewFile事件）时，就为该文件 buffer 定义一个自动命令， 该自动命令的意图是每当 CursorHold 事件触发（光标停留一段时间），就打印一个消息。

相当之下，<buffer> 参数更简单易懂，如该参数能满足局部自动命令的要求，优先使用这个吧。例如，将 :autocmd {event} <buffer> 命令放在某个函数内，先通过其他命令切换到正确的 buffer 内，再调用这个函数为该 buffer 定义局部自动命令。由于这已经是局部自动命令了，加不加组名的影响都不那么大了。

其他提示

- 自动命令是相对高级的功能，可用 has('autocmd') 判断你的 Vim 版本是否已编译 了这个功能，或 :version 看输出是否有 +autocmd。
- 文件类型检测的自动命令定义在 filetypedetect 组内，当你想创造新文件类型时，也可往这个组内添加自动命令，如 :autocmd filetypedetect *.xyx setfiletype xfile。但没事不要误用 :autocmd! 删除这个组内的其他自动命令。
- 嵌套的自动命令。默认情况下，自动命令中使用的命令如 :e :w 不再继续触发读写事件，但是加上 nested 可选参数，可允许嵌套。如 :autocmd {event} {pat} nested {cmd} 使得在执行 {cmd} 时有可能继续触发自动命令（不过有最大嵌套层数限制，除非必要，慎用）。nested 可选参数应位于 {cmd} 之前，只有保持 {cmd} 在最后部分，才方便在自动命令使用必要的空格啊。
- 自动命令也可以手动调用，当你觉得有这需求时再去查文档吧， :doautocmd 与 :doautoall。
- 太多自动命令有可能降低效率，因此有个选项 &eventignore 可以指定忽略某些事件。这不会删除自动命令，但有些事件不会触发了，相应自动命令也就不会执行了。在一个命令之前附加 :noautocmd {cmd} 可临时使得本次执行 {cmd} 时不会触发自动命 令。如 :noautocmd w 在这次写入过程中，不会触发写事件。

第三章 Vim 常用命令

3.6* 调试命令

对任何一门语言，都有必要掌握调试技巧或手段。本节介绍 VimL 语言编程可以怎么调试，介绍一些自己的经验与体会。

echo 大法

对于不太庞大的程序或脚本，在关键疑点处打印消息都是简单方便的发现问题的手段，姑且也算一种调试方法吧。

不过这明显有个问题，当程序调试完毕后，这些只为调试用的 echo 打印命令留着很碍事呀，可能会与正常的输出混杂在一起，干扰正常结果呢。所以最好是能将正常的 echo 与调试的临时 echo 区分开来。正好，VimL 有个奇葩规定，在每行行语句之前的 : 冒号是可选的。这是为了与命令行表观上一致，然而正常的 vim 脚本一般都不会自找麻烦多加这个冒号。但是若按语法规则，你在每行语句之前加一个冒号（甚至多个冒号）都是没有关系的。

于是，不妨自己规范一下，将调试用的打印语句，都写成 :echo，或者喜欢多个空格 : echo 也行。而在正常的程序输出语句中，则用整洁的无冒号 echo 版。这样，当调试完毕，确认程序无误后，就可以用 vim 强大的编辑命令将这些调试命令都删了：

```
: g/:\s*echo/delete
```

当然，你也许并不是想彻底删除，只是想注释掉，那就可用替换命令：

```
: g/:\s*echo/s/:\s*echo/" echo/
```

当 :s 命令使用的正则表达式与前面的 :g 命令的正则表达式是一样的时候，可以简写成 : g/:\s*echo/s//" echo/。因为 :s//{replace}/ 命令中，空模式的意图是重复使用上次的模式（寄存器 / 的内容）。若是为达这个目的，直接用替换命令也可以的：: %s/:\s*echo/" echo/。不过与 :g 命令联用（先查找目标行，再替换）会更灵活点，比如想将首列替换为注释符 "，而不影响内缩进的 :echo 命令，则可使用这样的替换命令：

```
: g/:\s*echo/s/^./"/
```

如果想更细致点，可以自行将 :echo 与 ::echo 用于不同场合，比如不同等级的调试输出。

还有个问题，:echo 命令的输出是易逝的，后一批的命令（vim 的解释单元）输出会覆盖掉前一批的命令输出。如果想保存这样的输入，有以下几种办法：

- :echomsg 用这个命令替换 :echo，则输出信息会保存在消息区，以后可用 :message 再次查看，当消息区的信息比较多时，可能需要翻页查看，G 跳到最后 一页，基本上就是最近的输出了。
- :redir 命令重定向，可以将随后的 :echo 消息重定向至文件、寄存器、变量中，当然也会同时显示在屏幕上。不再需要重定向功能时用 :redir END 命令取消。
- :redir! > {file} 重定向到文件中，当文件已存在时，用 ! 强制覆盖。
- ‘:redir @{reg}>’ 重定向至寄存器，如果支持系统剪贴板，用 * 或 + 表示。
- :redir => {var} 重定义向至一个变量中。
- :redir >> 将上述命令中的 > 换为 >> 表示附加。
- &verbosfile 将详情信息写入这个选项值指定的文件中。&verbose 选项值设定详 情信息的等级。

断点进入调试模式

Vim 也提供了正式的调试模式，那有点像允许单步执行的 Ex 模式。一般需要先设置断点，随后当脚本运行到断点处，就进入了调试模式。添加断点用 :breakadd 命令：

- :breakadd file [lnum] {name} 在一个 vim 脚本文件中的某行加断点，行号可选。注意如果提供行号，行号参数位于文件名之前，如果省略行号，相当于第 1 行。随后 当 :source {name} 加载该脚本时，执行到那行时会暂停，进入调试模式。
- :breakadd func [lnum] {name} 在某函数的第几行打断点。{name} 指函数名。如 果是全局函数，那就是直接的函数名，如 FuncName。如果是脚本局部函数，如 s:FuncName ，则要先找到那个脚本在当前 vim 会话的脚本号（:scriptnames），然后实际的函数名是 <SNR>dd_FuncName，其中 dd 就是脚本号数字。如果是匿名函数，它没有名字，就只能用其函数编号了，如 :breakadd func 1 21 表示在第 21 个匿名函数的第1行处打断点。那匿名函数编号如何确定呢？如果这个函数有出错了，在错误信息中会打印出出错函数的名字与行号，匿名函数没名字就用编号代替了。（没有出错么？没出错为啥调试？）至于 [lnum] 行号，可理解为函数体内相对于函数头定义的相对行号，可不是该函数定义块在脚本文件中的行号。即从函数定义头按 [lnum] 次 j 就是函数断点处。
- :breakadd here 当你在编辑一个 vim 脚本文件时，相当于在当前文件的当前行加入断点。如果你已经进入了调试模式，并且已经单步进入了某个函数，:breakadd here 也可以在当前函数的当前行加入断点，下次再次调用该函数时（或下次循环）运行到此处时也会暂停。

当用 :breakadd 添加了一些断点后，可用 :breaklist 查看断点信息。也可用 :breakdel 删除断点。

- :breakdel {nr} 按断点号删除某个断点（:breaklist 会列出断点号）。
- :breakdel * 删除所有断点。
- :breakdel file [lnum] {name}
- :breakdel func [lnum] {name}
- :breakdel here 这三个命令与 :breakadd 相似，但是删除断点。

除了通过 :breakadd 添加断点，以期将来运行到彼处时进入调试模式外，还有另外两种方式直接进入调试模式：

- :debug {cmd} 在执行命令之前附加 :debug ，就将在执行该命令时立即进入调试模式，一般接着用 s （step in）深入调试，如果用 n （step over）可能就将整条 {cmd} 命令当作一步直接执行完了，并不能达到调试效果。
- vim -D {other args} 在启动 vim 时，通过 -D 命令行参数，直接在加载 vimrc 时就开始进入调试模式了。

调试模式

调试模式是一种特殊的 Ex 模式，除了一般的 ex 命令，还可以使用以下调试命令：

- cont (c)，表示继续执行，直到遇到下一断点，或结束。
- quit (q)，中断，类似 <Ctrl-C>
- interrupt (i)，也类似 <Ctrl-C>

- next (n)，单步执行，类似 step over，会跳过函数调用与加载文件。
- step (s)，单独执行，类似 step in，会步进函数调用或加载文件。
- finish (f)，结束当前加载脚本或函数调用，回到调用处。
- backtrace (bt) 或 where，显示调用堆栈。
- frame (fr) {N} ，切换到堆栈的第 N 层，可用 + - 表示相对层。
- up / donw, 在堆栈处上移一层（fr +1）或下移一层（fr -1）。

以上这些调试命令可以尽可能缩写，只要前缀字符不冲突（小括号里也已标出最简缩写）。直接敲回车表示重复上一次命令，这样就不必每次输入 s 或 n 命令了。

调试命令没有补全功能，只有普通 ex 命令才能补全。如果要使用与调试命令相同的普通 ex 命令，多加一个冒号，如 :next。但是，由于在 Ex 模式，编辑窗口是不更新的（事实上，只要调试过程稍长，vim 窗口就完全被调试信息覆盖了），很多普通 ex 命令是 没有效果后，只有在完成调试模式后重回普通模式才能反映编辑窗口的变化。

真正有价值的 ex 命令是可用 echo 命令查看变量值，并且能根据当前环境查看相应作用域的变量值，比如在加载脚本时可查看 s:var，运行到函数内部可看局部变量 l:var （在函数内默认局部变量，:echo var 就相当于 :echo l:var）。而在正常的命令行下面，是无法查看 s:var 与 l:var 变量的。

在调试模式中，只能打印出正要执行的那行的源代码。这是典型的命令行式的调试方式，并不能像 IDE 那般分裂出源码窗口，直接将光标定位到正在执行的行上。如果想查看完整代码，只能用另外一个 vim 打开源文件查看了（有可能出现 *.swp 冲突问题，用只读模式打开就好）。所以 VimL 调试的可视化程序仍稍嫌不足，希望日后还有改进。

第四章 VimL 数据结构进阶

在第 2.1 章已经介绍了 VimL 的变量与类型的基本概念。本章将对变量类型所指代的数据结构作进一步的讨论。

4.1 再谈列表与字符串

引用与实体

前文讲到，列表作为一种集合变量，与标量变量（数字或字符串）有着本质的区别。其中首要理解的就是一个列表变量只是某个列表实体的引用。

直接用示例说话吧，先看数字变量与字符串变量的平凡例子：

```
: 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
```

我们先创建了一个数字变量 x，并为其赋值为 1，然后再创建一个变量 y，并为 x 的值赋给它。显然，现在 x 与 y 的值都为 1。随后我们改变 y 的值，重赋为 2，再查看两个变量的值，发现只有变量 y 的值改变了，x 的值是没改变的。因此，即使在创建 y 变量时用 :let y = x 看似将它与 x 关联了，但这两个变量终究是两个独立不同的变量，唯一有关联的也不外是 y 初始化时获取了 x 的值。 此后这两个变量分道扬镳，可分别独立地改变运作。对于字符串变量 a 与 b，也是 这个过程。

然后再看看列表变量：

```
: let aList = ['a', 'aa', 'aaa']
: let bList = aList
: echo 'aList:' aList 'bList:' bList
: let bList = ['b', 'bb', 'bbb']
: echo 'aList:' aList 'bList:' bList
```

结果似乎与上面的数字或字符中标题很相似，没什么差别嘛。虽然 bList 一开始与 aList 表示同一个变量，但后来给 bList 重新定义了一个列表，也没有改变原来的 aList 列表。这与字符串 a b 的关系很一致呢。

但是，我们重新看下面这个例子：

```
: 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
```

这里先把原来的 aList bList 变量删除了，以免上例的影响。仍然创建了列量变量 aList，与 bList 并让它们“相等”。然后我们通过 bList 变量将列表的第一项 [0] 改成另一个值 b，再查看两个列表的值。这时发现 aList 列表也改变了，与 bList 作出了同样的改变，两者仍是“相等”。

通过这组试验，想说明的是，当 VimL 创建一个列表（变量）时，它其实是在内部维护了一个列表实体，然后这个变量只是这个列表实体的引用。命令 :let aList = ['a', 'aa', 'aaa'] 相当于分以下两步执行工作：

1. new 列表实体 = [‘a’, ‘aa’, ‘aaa’]
2. let aList = 列表实体的引用

然后命令 :let bList = aList，它只是将 aList 变量对其列表实体的引用再赋值给变量 bList，结果就是，这两个变量都引用了同一个列表实体，或说指向了同一个列表实体。而命令 :let bList[0] = 'b' 则表示通过变量 bList 修改了它所引用的列表的第一个元素。但变量 aList 也引用这个列表实体，所以再次查看 aList 时，发现 它的第一个元素也变成 'b' 了。实际上，不管是对 aList 还是 bList 进行索引 操作，都是对同一个它们所引用的那个列表实体进行操作，那是无差别的。

对于普通标量变量，则是另一种情况。当执行命令 :let b = a 时，变量 b

就已经 与 a 是无关的两个独立变量，它只是将 a 的值取出来并赋给 b 而已。但 :let bList = aList 是将它们指向同一个列表实体，在用户使用层面上，可以认为它们是同一个东西。但是当执行 :let bList = ['b', 'bb', 'bbb'] 后，变量 bList 就指向另一个列表实体了，它与 aList 就再无联系了。

可见，当对列表变量 bList 进行整体赋值时，就改变了该变量所代表的意义。这时与对字符串变量 b 整体赋值是一样的意义。然而，标量始终只能当作一个完整独立的值使用，它再无内部结构。例如，无法使用 let b[0] = 'c' 来改变字符串的第一个字符，只能将另一个字符串整体赋给 b 而达到改变 b 的目的。

总结，只要牢记以下两条准则：

- 标量变量保存的是值；
- 列表变量保存的是引用。

函数参数与引用

我们再通过函数调用参数来进一步说明列表的引用特性。

举个简单的例子，交换两个值，可以引入一个临时变量，由三条语句完成：

```
: let tmp = a
: let a = b
: let b = tmp
```

这种交换值的需求挺常见的，考虑包装成一个函数如何？

```
: function! Swap(iValue, jValue) abort
:     let l:tmp = a:iValue
:     let a:iValue = a:jValue
:     let a:jValue = l:tmp
: endfunction
```

但是，当尝试调用 :call Swap(a, b) 时，vim 报错了。因为参数作用域 a: 是只读 变量，所以不能给 a:iValue 或 a:jValue 赋另外的值。但是，即使参数不是只读的，这样的交换函数也是没效果的（比如用 C 或 python 改写这个交换函数）。因为在调用 Swap(a, b) 时，相当于先执行以下两个赋值语句给参数赋值：

```
: let a:iValue = a
: let a:jValue = b
```

此外，不管在函数内不管怎么倒腾参数 a:iValue 与 b:jValue，都不会影响原来的 a 与 b 变量。因为如前所述，标量赋值，只是拷贝了值，等号两边的变量是再无联系的。

但是，交换列表不同位置上的元素是可实现的，比如把上面那个交换函数改成三参数版，第一个参数是列表，跟着两个索引：

```
: function! Swap(list, idx, jdx) abort
:     let l:tmp = a:list[a:idx]
:     let a:list[a:idx] = a:list[a:jdx]
:     let a:list[a:jdx] = l:tmp
```

```
: endfunction
```

请试运行以下语句确认这个函数的有效性：

```
: echo aList
: call Swap(aList, 0, 1)
: echo aList
```

在写较复杂的 VimL 函数时，一般不建议在函数体内大量使用 a: 作用域参数。因为传入的参数是无类型的，很可能是不安全的。最好在函数的开始作一些检查，合法后再将 a: 参数赋给一个 l: 变量，然后在函数主体中只对该局部变量操作。此后，如果能参数的假设需求有变动，就只在修改函数前面几行就可以了。例如再将交换函数改成如下版本：

```
: function! Swap(list, idx, jdx) abort
:     if type(a:list) == v:t_list || type(a:list) == v:t_dict
:         let list = a:list
:     else
:         return "
:     endif
:
:     let i = a:idx + 0 "
:     let j = a:jdx + 0
:
:     let l:tmp = list[i]
:     let list[i] = list[j]
:     let list[j] = l:tmp
: endfunction
```

再用以下语句来测试修改版的交换函数：

```
: call Swap(aList, 1, 2)
: echo aList
```

可见，即使在函数体内，将参数 a:list 赋给另一个局部变量 l:list，交换工作也正常运行。因为 g:aList a:list 与 l:list 其实都是同一个列表实体的引用啊。

列表解包

在 3.4 节我们用 execute 定义了一个 :LET 命令，用于实现连等号赋值。但实际上可以直接用列表赋值的办法实现类似的效果。例如：

```
: LET x=y=z=1
: let [x, y, z] = [1, 1, 1]
: let [x, y, z] = [1, 2, 3]
```

其中前两个语句的结果完全一样，都是为 x y z 三个变量赋值为 1。注意等号左边也需要用中括号把待赋值变量括起来，分别用等号右侧的列表元素赋值。这种行为就叫做列表解包（List unpack），即相当于把列表元素提取出来放在独立的变量中。显然用这种方法为多个变量赋值更具灵活性，可以为不同变量赋不同的值。

这个语法除了可多重赋值外，还能方便地实现变量交换，如：

```
: let [x, y] = [y, x]
```

用过 python 的对此用途应该很有亲切感。不过在 VimL 中，等号两边的中括号不可省略，且等号两边的列表元素个数必需相同，否则会出错。不过在左值列表中可以用分号分隔最后一个变量，用于接收右值列表的剩余元素，如：

```
: let [v1, v2; rest] = list
"
: let v1 = list[0]
: let v2 = list[1]
: let rest = list[2:]
```

在上例中假设 list 列表元素只包含简单标量，则解包赋值后，v1 v2 都是只接收了一个元素值的标量，而 rest 则接收了剩余元素，它还是个（稍短的）列表变量。而 list[2:] 的语法是列表切片（slice）。

索引与切片

这里再归纳一下列表的索引用法：

- 索引从 0 开始，不是从 1 开始。
- 可以使用负索引，-1 表示最后一个索引。
- 可以使用多个索引，这也叫切片，表示列表的一部分。

要索引一个列表元素时，用正索引或负索引等效的，这取决于应用场合用哪个方便。如果列表长度是 n，则以下表示法等效：

```
list[n-1] == list[-1]
list[0]   == list[-n]
list[i]   == list[i-n]
```

然而，不管正索引，还是负索引，都不能超出列表索引（长度）范围。

列表切片（slice）是指用两个索引提取一段子列表。list[i:j] 表示从索引 i 到索 引 j 之间（包含两端）的元素组成的子列表。注意以下几点：

- i j 同样支持负索引，不管用正负索引，如果 i 索引在 j 索引之后，则切片结果是空列表。
- 如果 i 超出了列表左端（0 或 -n），或 j 超出列表右端，结果也是空列表。
- 可省略起始索引 i，则默认起索引为 0；省略结束索引 j，则默认是最后一个索引 -1；如果都省略，只剩一个冒号，list[:] 与原列表 list 是一样的（但是另一个拷贝列表）。
- 可以为切片赋值，即将一个列表的切片放在等号左边作为左值，可改变索引范围内的元素值，但一般右值要求是与切片具有相同项数的列表。
- 不支持三索引表示步长，list[i:j:step] 或 list[i:j:step] 在 VimL 中是非法 的，不支持跳格切片，只支持连续切片。
- list[s:e] 表示法有歧义，因为可能存在脚本局部变量 s:e，则用该变量值单索引列表。可在冒号前后加空格避免歧义，list[s : e] 表示切片。

处理列表的内置函数

VimL 提供了一些基本的内置函数用于列表的常用操作，详细用法请参考文档 `:help list-functions`，这里仅归纳概要。

- 查询列表信息的函数：
- len(list) 取列表长度，列表的最大索引是 len(list)-1。
- empty(list) 判断列表是否为空，即列表长度为 0。
- get(list, i) 相当于 list[i]，但是当 i 超出索引范围时，get() 函数不会出错，且 可再提供第三参数表示超出索引时的默认值（如果省略，默认值0）。
- index(list, item) 查找一个元素在列表中的位置，如果不存在该元素，则返回 -1。
- count(list, item) 检查一个元素在列表中出现多少次。
- max(list) min(list) 查询一个列表中的最大或最小元素。
- string(list) 将列表转化为字符串表示法。
- join(list, sep) 将列表中的元素用指定分隔符连接为一个字符串表示。
- 修改列表元素的函数：
- add(list, item) 在列表末尾添加一个元素。
- insert(list, item) 在列表头部添加一个元素，比 add() 尾添加低效。但 insert() 可额外提供第三参数表示要插入的索引位置，省略即 0 表示插在最前面。
- remove(list, idx) 删除位置 idx 上的一个元素，remove(list, i, j) 删除从 i 到 j 索引之间的所有无素，相当于 unlet list[i:j]。
- 生成列表的函数：
- range() 支持一至三个参数，生成连续或定步长的数字列表。
- extend(list1, list2) 连接两个列表，相当于 list1+list2，但 extend 会原位修改 list1 列表。与 add() 函数不同的是，add 只增加一个元素，而 extend 是加入另一 个列表。
- repeat(list, count) 相当于不断连接自身，总计重复 count 次，生成一个更长的列表。
- copy(list)，生成一个列表副本，用等号赋值只是引用同一个列表实体，用 copy() 函 数才能生成另一个新列表（每个元素值与原列表相同而已）。copy() 函数是浅拷贝， 列表元素直接赋值。如果要考虑列表元素也可能是列表或字典（引用），则用 deepcopy(list) 递归拷贝完全的副本。
- reverse(list) 将一个列表倒序排列，原位修改原列表。
- split(list, pattern)，将一个字符串分解为列表，相当于 join() 的反函数。

- 分析列表的高阶函数：
- sort(list) 为一个列表排序。
- uniq(list) 删除列表中相邻的重复元素，列表需已排序。
- map(list, expr) 将列表每个元素进行某种运算，将结果替换原元素。
- filter(list, expr) 将列表每个元素进行某种运算，若结果为 0，则删除相应元素。

这些高阶函数，除了都会原位修改作为第一个参数的列表外，都还能接收额外参数表明如何处理每个元素。由于额外参数可以是另一个函数（引用），所以称之为高阶函数。其具体用法略复杂，在后面相关章节将继续讲解部分示例。

字符串与列表的关系

字符串在很大程序上可以理解为字符列表，可以用类似的索引与切片机制。但是，字符串与列表的最大区别在于，字符串是一个完整的不可变标量。所以，凡是可以改变列表内部某个元素的操作（如索引赋值、切片赋值）或函数（如 add/remove 等），都不可作用于字符串。而 copy() 也没必要用于字符串，直接用等号赋值即可。不过 repeat() 函数作 用于字符串很有用，能方便生成长字符串。

将字符串打散为字符数组，可用如下函数方法：

```
: let string = 'abcdefg'
: let list = split(string, '\zs')
: echo list
```

split(string, pattern) 函数是将字符串按某种模式分隔成列表的。`\zs` 不过是一种 特殊模式，它可以匹配任意字符之间（详情请参考正则表达式文档），所以结果就是将每个字符分隔到列表中了。

第四章 VimL 数据结构进阶

4.2 通用的字典结构

为什么说字典是通用结构。因为在 VimL 中字典是最复杂的内置类型了，而更复杂的数据结构都能以字典为基础构建出来。不过从基础的概念上理解，字典与列表其实也有些相似之处，当掌握了列表之后，对字典的用法也就容易了。

字典与列表的异同

在其他一些（脚本）语言中，对应 VimL 的列表与字典的概念，也叫数组与关联数组。所以字典也可以看成是一种特殊的列表，无序的以字符串为索引的列表。字典的索引也叫键，在 VimL 中，字典的键只能是字符串，当数字用作字典键时也被隐式转为字符串。其实类型的值，一般不能用作字典的键。

请看这个示例：

```
: 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() 创建了一个列表 list，包含的元素是 0-9 这十个数字。然后创建了一个空字典 dict，再用循环为字典增加键值，也用相同的 0-9 这十个数字作为键与值。这样，在表观上，dict 与 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] 与 list[5] 都得到数值 5。但通过 :echo dict 可以发现，dict 字典的键，其实不是数字，而是字符串（'0', '1'等）。

为理解遍历字典的范式 for [k, v] in items(dict)，可先用 echo items(dict) 查 看这是什么。可见 items(dict) 返回一个列表，该列表的每个元素又是个小列表，包含键与值两个元素。所以你明白了，字典的元素，不像列表的元素那么简单的一个值，而是一个“键值”对。所谓关联数组名称也源于此，每个键对应一个值。键是唯一的，但值可不唯一，即不同键可关联相同的值。

在字典循环中，也用到了上节介绍的列表解包的多重赋值的功能，相当于如下语句：

```
: let [k, v] = items(dict)[0]
: let [k, v] = items(dict)[1]
: ...
: let [k, v] = items(dict)[9]
```

也因此，[k, v] 必须用中括号括起来。

在这个特殊的例子中，遍历字典所得的值也许是与列表一样有序。但请记住，字典不保证有序。同时，在一般应用中，最好不要用连续的数字作为字典的键，那应该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，则用字典或许是有意义的。如：

```
: let dict[10] = 'a'
: let dict[100] = 'b'
: let dict[1000] = 'c'
```

这样，只为 dict 增加了三个元素。但若为 list[1000]='c' 赋值，则会为列表增加 1000 个元素，中间的无用索引都浪费了。

在常用使用字典时，建议用简单字符串索引，所谓简单字符串，即是可充当 VimL 标记符 （变量名）的字符串。这时，字典的中括号索引可用点索引简化写法，即 dict['name'] 可简化等效于 dict.name。当索引是一个字符串变量时，用中括号索引更方便，即 dict[varname]。

还有一点需要重点理解的是，字典变量与列表变量一样，是引用而已。请看以下示例：

```
: let d1 = {}
```

```
: let d2 = {}
: echo d1 == d2
: echo d1 is d2
: let d3 = d1
: echo d3 is d1
```

虽然 d1 与 d2 都是空字典，它们按值比较是一样的，但其实是不同的字典实体，用 is 比较显示不一样。为另一个变量 d3 赋值后，就指向相同的字典实体了。

操作字典的内置函数

上节介绍的许多关于列表的函数，也可作用于字典。只是其他参数意义可能不一样，对于字典时，一般是根据键来处理的。详情请查阅 :h dict-functions。

但以下几个函数是字典特有的：

- has_key(dict, key) 检查一个字典是否含有某个键。
- keys(dict) 返回由字典的所有键组成的列表。
- values(dict) 返回由字典的所有值组成的列表。
- items(dict) 返回由字典的所有键值对组成的列表。

这几个返回列表的函数一般用于 for ... in 循环中。字典的内部存储是无序的，但可用 for ... in sort(keys(dict)) 根据键顺序遍字典。

第四章 VimL 数据结构进阶

4.3 嵌套组合与扩展

VimL 虽然只提供了列表与字典两种数据结构，但通过列表与字典的合理组合，几乎能表达任意复杂的数据结构。这与许多其他流行的脚本语言（如 python）的思想如出一辙。本节就讨论在 VimL 中如何用列表与字典表示常用数据结构。

堆栈与队列

堆栈是所谓先进后出的结构，队列是先进先出的结构。这可以直接用一个 list 表示， 因为 list 相当于个动态数组，支持随意在两端增删元素。

如果只在列表尾部增删元素，那就实现了堆栈行为。如果尾部增加而在头部删除，就实现了队列行为，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
```

在这个示例中，用 Push/Pop 表示堆栈操作，用 Push/Shift 表示队列操作。这只为简明地说明算法意图，实际应用中最好先检查 stack/queue 是否为 list 类型，以 及检查列表是否为空。

链表

在脚本语言中，其实根本不用实现链表，因为动态数组本身就可用于需要链表的场合。在 VimL 中，就直接用 list 表示线性链就够了。除非你真的需要很频繁地在一个很长的 list 中部增删元素，那么或可用字典来模拟链表的实现。

例如，以下代码构建了一个有 10 个结点的链表，每个结点是个字典，value 键表示存储 内容，next 表示指向下一个结点：

```
: let head = {}
: for value in range(10)
:   let node = {'value': value, 'next': head}
:   let head = node
: endfor
```

其实在上面的循环中，临时变量 node 可以省略。head 始终指向链表的起始结点，可通过 next 键依次访问剩余结点，末尾结点的 next 键值是空字典。

这里的关键是，字典的值，或列表元素的值，不仅可以存储像数字与字串符的简单标量，还可以存储另一个列表或字典（的引用）。基于这样的嵌套与组合，就可以表达更复杂的数据结构了。

二维数组（矩阵）

如果列表的每个元素都是另一个列表，那就构成了一个二维数组。例如：

```
: let matrix = []
: for _ in range(10)
:   let row = range(10)
:   call add(matrix, row)
: endfor
```

构建了一个 10x10 大小的矩阵，其中每个行向量由 range(10) 生成。这样快速生成的矩阵每一行都相同，或许不是很有趣，但是可以用以下两层循环重新赋值：

```
: for i in range(10)
:   for j in range(10)
:     let matrix[i][j] = i * j
:   endfor
: endfor
```

从数学意义上的矩阵讲，它应是规整的矩形，即每行的长度是一样的。但当在 VimL 中用 列表的列表表示时，其实并不能保证每一行都等长。例如：

```
: let text = getline(1, '$')
: for i in range(len(text))
:   let line = text[i]
:   let text[i] = split(line, '\s\+')
: endfor
```

在这里，首先用 getline() 获取当前 buffer 的所有行，保存在 text 这个列表变量中，其中每个元素表示一行文本字符串。在随后的循环中，又将每行文本分隔成一个个单词（空格分隔的字符串），将标量字符串元素转化为了另一个列表。因此，text 最 终结果就是列表的列表，即二维数组。而一般情况下，每行的单词数量是不等，所以这个二维数组不是规整的矩阵。

事实上，这个示例的循环可以直接用 map() 函数代替：

```
: let text = getline(1, '$')
: call map(text, "split(v:val, '\\s\\+')")
```

树

以二叉树为例，也可用一个字典来表示树中某结点，除了需要一个键（如 value）来保存业务数据，还用一个 left 键表示左孩子结点，right 表示右孩子结点，这两个应该都是另一个具有相同结构的字典引用，如果缺失某个孩子，则可用空字典表示。

```
: let node = {}
: let node.value = 0
: let node.left = {}
: let node.right = {}
```

这样，只要有一个字典变量引用了这样的一个结点（不妨称之为根结点），就相当于引用着一棵树，沿着结点的 left 与 right 键就能访问整棵树的所有结点。两个子结点都是空字典时，该结点就是所谓的叶结点。

不过，由于每个结点含有两个方向的子结点，要遍历树可不是那么直观。有兴趣的读者请参考相应的树算法。本节内容旨在说明 VimL 的字典用法，展示其表达能力。而算法其实是与语言无关的。

在上述的树结点字典结构中，只能从一个结点访问其子结点，而无法从子结点访问父结点。如果有这个需求，只要在每个结点字典中再加一个键引用父结点即可，如：

```
: let node.parent = {}
```

每个子结点都有父结点，即 parent 键非空。根结点没有父结点，那 parent 键应该 存个什么值呢？可以就用空字典表示，也可以引用它自身，这都可以将根结点与其他非根结点区分开来。

我们知道，字典或列表变量都只是某个实体的引用。VimL 的自动垃圾回收机制主要是基于计数引用的。如果某个字典或列表实体没有被任何变量引用了，即引用计数为 0 时， （在变量离开作用域或显式 :unlet 时会减少引用计数）VimL 就认定该实体无法被访问了，就会当作垃圾回收其所占用的内存。在大部分简单场合中，这套机制很好用。不过考虑这里讨论的包含 parent 与 left right 键的树结点，在父、子结点之间形成了环引用，它们的引用计数始终不会降到 0 。然而 VimL 另外也有一个算法检测环引用

，所以也尽可放心使用这个树结构，不必担心内在泄漏。只不过存在环引用时，垃圾回收的时机可能相对滞后而已。

现在，让我们再考虑一种有任意多个孩子的树（任意叉树）。这种结构在实际应用中是存在的，比如目录树，每个目录（结点）可以有很多个不确实数量的子目录或文件（叶结点）。为表示这种结构，我们可以将所有子结点放在一个列表中，然后用一个键引用这个列表，如下定义每个结点的字典结构：

```
: let node = {}
: let node.value = 0
: let node.parent = {}
: let node.child = []
```

与原来的二叉树相比，取消 left 与 right 键，而以统一的 child 键代替。每当增加一个子结点时，就添加到 child 列表中，同时维护该子结点的 parent 键。如 果 child 键为空列表，就表示该结点为叶结点。

图

图是一些顶点与边的集合，常用 G(V, E) 表示，其中 V 是顶点集合，E 是边集合，每条边连接着 V 中两个顶点。一般用 |V| 表示顶点的个数，|E| 表示边数。

用程序表示图，有两种常用的方式，邻接矩阵与邻接表。这里讨论一下如何用 VimL 的数 据结构表示图。

邻接矩阵很简单，就是一个 |V| x |V| 大小的矩阵，假设就用变量名 graph 表示这 个矩阵。前面小节已介绍，矩阵在 VimL 中就是列表的列表。如果顶点 i 与 j 之间 有一条边，就 :let graph[i][j] = 1，否则就用一个特殊值来表示这两个顶点之间没有边，比如在很多情况下用 0 表示无边是可行的，:let grapsh[i][j] = 0。如果是 有权边，则可把边的权重保存在相应的矩阵位置中，如 :let graph[i][j] = w。如果 是无向图，则再对称赋值 :let graph[j][i] = graph[i][j]。

由于矩阵元素支持随机访问，用邻接矩阵表示图在某些应用中非常高效简便，尤其中边数非常稠密的情况下（极限情况是每两个顶点之间都有边的完全连通图）。不过在边数很少的情况下，这将是个稀疏矩阵，在内存空间使用上比较低效。

邻接表，首先它是包含所有顶点的列表；每个顶点是一个字典结构，它至少有个键 edge 来保存所有与本顶点相关的边，这应是一个边结构的列表；在边结构字典中则保存着权重 weight，以及它所连接的顶点（字典引用）。大致结构如下所示：

```
: 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访问边结构，那这个字典中可以只保存一个顶点，另一个顶点就是它被保存的顶点（由它的 edge 键访问到这个边）。这可以减少一些存储空间，不过顶点也只是字典引用，保存双端点也浪费不了太多空间。

在实际的图应用中，肯定还会有具体的业务数据，这些数据一般是保存顶点结构中。比如可以给每个顶点给个 id 编号或名字，如果有大量复杂的数据，可单独保存在另一个字

典引用中。

所以，邻接表虽然复杂，但灵活度高，易扩展业务数据。而邻接矩阵在矩阵元素中只能保存一个值，扩展有些不方便。除非是业务数据是保存在边结构中，那么在矩阵中可以保存另一个字典引用，而不是简单的权重数值。

JSON

如果你了解 JSON，就会发现 VimL 的列表与字典的语法表示，正好也是符合 JSON 标准 的。一个有效的 JSON 字符串也是也是合适的 VimL 的表达式，可以直接用于 `:let` 命 令的赋值。

当然这有一个小小的限制，JSON 字符串不能有换行，因为 VimL 语言是按行解析的，且续行符比较特殊（在下一行开头使用反斜杠）。如果是不太复杂的 JSON，在 Vim 编辑中 可以将普通命令 `J` 将多行字符串合并为一行，我不认为你会用其他编辑器写 VimL 脚 本。

此外，有个内置函数 `jsondecode()` 可将一个合法的 JSON 字符串（允许多行）解析为 VimL 值，以及反函数 `jsonencode()` 将一个 VimL 表达式转换为 JSON 字符串。

总结

在 VimL 中，用列表与字典的组合，可以表达很复杂很精妙的数组结构，几乎只有想不到没有做不到。其实这也不必奇怪，因为目前大部分作为高级语言的动态脚本，其思想是相通的。虽然 VimL 似乎只能用于 Vim，但它与其他流行的外部脚本语言，在某种程序上是极其相似的。

第四章 VimL 数据结构进阶

4.4* 正则表达式

在本章末尾，再简要介绍一下正则表达式的内容。正则表达式对于 Vim 很重要，但本教程不打处专门用一章的内容来讲叙正则表达式（实际上正则表达式的内容可以写一本书）。插录在这章数据结构之后，你可以认为正则表达式也是一种表达字符串内部结构的模糊方法——模糊其实比精确更难理解与掌握。

Vim 对于正则表达式的内置帮助文件请查阅 `:h pattern.txt`。

Vim 正则表达式的设置模式

很多编程语言或工具软件，都支持正则表达式，所以这是一种很实用的通用技能。然而不幸的是各家支持的正则表达式都“略有不同”，更不幸的是 Vim 自家里面还有几种不同的正则表达式语法，这是通过选项设置 `&magic` 改变正则表达式“包装套餐”的。

Vim 一共支持四套正则表达式，在 `/` 或 `?` 命令行中可添加特殊前导字符来表示本次搜索采用哪套正则表达式：

- `\v`(very magic)，最接近 `perl` 语言的正则表达式。除了常规标识符字符外，大多数字符都有特殊含义，即魔法字符。

- \m(magic)，这是 Vim 的标准正则表达式。主要特征是括号与加号都是字面意义，不是魔法字符，需要在前面多加一个反斜杠来表示魔法意义。
- \M(nomagic) 更少的魔法字符，点号(.)与星号(*)都是字面意义。
- \V(very nomgic) 只有斜杠本身及正则表达式定界符有特殊意义，其他所有字符按字面意义匹配。

这四种正式表达式是根据魔法字符的多寡程度划分的。但是用反斜框可以改变魔法字符的意义。即在一种正则表达式中，如果一个字符是魔法字符，反斜杠转义后就表示字面意义；反之如果一个字符不是魔法字符，加反斜杠转义后就可能成为魔法字符表示特殊意义。例如在 \m 正则表达式中，加号 + 不是魔法字符，它匹配字面的加号，使用 \+ 表示匹配前面那个字符一次以上。而在 \v 正则表达式中，+ 是魔法字符，表示匹配前面那个字符一次以上，而用 \+ 匹配字面上的加号。

如果没有显式指定哪种正则表达式（这应该是大部分 vimer 使用 / 搜索的默认方式），就根据 &magic 选项决定。设定了 :set magic 就默认使用 \m 正则表达式，设定 :set nomagic 就默认使用 \M 正则表达式。但是若要使用 \v 或 \V 必须显示指定。因为 &magic 选项的默认值是开启的，所以 Vim 的默认正则表达式是 \m 这套，不妨称之为 Vim 的标准正则表达式。

为什么正则表达式已经很复杂了，Vim 还要增加几种非标准正则表达式来使之更复杂？我想这是 Vim 的另一个设计原则：尽可能减少用户的手动输入字符数（击键次数）。Vim 是一个通用文本编辑器，所编辑的文件内容在不同场合或有不同的侧重。比如在编辑程序源文件时，应该普遍会有很多括号，很可能就需要经常搜索字面意义的括号，这时用标准的 \m 正则表达式就更方便，而用 \v perl 类的正则表达式，就必须用 \(\) 来搜索一对空括号。而在另外一些场合，可能希望直接用 () 来表示组合，这就用 \v 正则表达式更方便了。

对于精通（或习惯）perl 类正则表达式的用户，可以通过简单映射 :nnoremap // /\v 自动添加 \v 前缀，始终使用 perl 类正则表达来搜索。在替换命令 :s/// 的模式 部分，也可以添加 \v 或其他前缀显式指定正则表达式的标准。

但是，对于一些需要正则表达式作为参数的内置函数，如 match()，只使用 magic 的标准正则表达式。这可能主要是考虑函数实现的方便与效率吧。毕竟函数主要写在 VimL 脚本中，而脚本一般只需写一次，语义一致也更重要。

因此，对于 Vimer 用户，还是建议掌握 Vim 的标准正则表达式。对于其他三种非标准正则表达式，了解就要，觉得方便有用时，尽管一试。本文剩余部分只介绍 Vim 标准正则 表达式的基本语法。

Vim 标准正则表达式

正则表达式描述的是如何匹配一个字符串，简单地说，它试图说明以下几个基本问题：

- 匹配什么字符
- 匹配多少次
- 在哪里匹配（定位限制）

再高级的议题还有分组与前向自引用等。

匹配字面字符

以下字符按字面意义匹配（非魔法字符）：

- 大小字符与小写字母：A-Z a-z
- 数字：0-9
- 下划线：_
- 加号：+
- 竖线：|
- 小括号与大括号：() {}
- 其他没有定义特殊意义的符号，以及其他指明要加反斜杠转义才表示特殊意义的字符。

匹配字符类别

支持的常用字符类别有：

- \s：空白字符
- \d：数字字符（0-9）
- \w：单词字符（合法标识符）
- \h：合法标识符的开头
- \a：字母
- \l：小写字母
- \u：大写字母

以上这几类字符表示，若改用大写，则表示取反，如 \S 表示非空白字符。这与大多数正则表达式的语法表示是一致的。

Vim 正则表达式还有几类字符表示与选项相关，由相应选项指定字符集。

- \i 由选项 &isident 指定的标识符
- \k 由选项 &iskeword 指定的关键字符
- \f 由选项 &isfname 指定的可用于文件名（路径）的字符
- \p 由选项 &isprint 指定的可打印字符
- \I \K \F \P 大写版本在以上小写版本基础上排除数字

也可以手动指定字符范围，用中括号 []：

- [] 匹配括号内任意一个字符，如 [abcXYZ] 可匹配这六个字符中的作一个
- [0-9] 用短横线（减号）指定的连续字符范围，匹配该范围内任一字符
- [^0-9] 匹配非数字，括号内第一字符是 ^ 时表示取反
- [-a-z] 若要包含减号本身，放在中括号内第一个字符，该示例表示小写字母或减号

中括号的用法与其他多数正则表达式一样。所以中括号与大小括号不一样，它是魔法字符，若要匹配字面的中括号，则须用 \[或 \]。

Vim 正则表达式还支持另一种特殊的中括号用法：

- \%[] 匹配中括号内可选的连续字符串，类似 ex 命令的缩写语法。

例如 :edit 可缩写至 :e，用正则表示就是 :e\%[dit]。

其他一些特殊字符：

- . 任一单字符
- \t 制表符
- \n 换行符
- \r 回车符
- \e <Esc> 键

匹配重复多次

- 0 或多次：*
- 1 或多次：\+
- 0 或 1 次：\? 或 \=
- 指定次数范围：\{n,m}，右大括号不需要反斜杠转义，左大括号需转义
- 非贪婪的次数范围：\{-n,m}

所以，没有意外地，点号与星号是魔法字符，分别用于匹配任意字符与任意次数，加反斜杠匹配字面点号（\.）与星号（*）。但是问号 ? 不是魔法字符，须用 \? 来表示匹配 0 或 1 次。

这些语法项不能单独使用，须用于表示字符（或类别）的后面，表示匹配前面那个字符字符多少次。\{n,m} 是通用的次数表示语法，可以省略 n 或（与）m。

- \{n} 严格匹配 n 次
- \{n,} 至少要匹配 n 次
- \{,m} 匹配 0 至 m 次
- \{} 匹配 0 或多次，等同于 *
- \{0,1} 匹配 0 或 1 次，等同于 \?
- \{1,} 匹配 1 或多次，等同于 \+

正则表达式一般采用贪婪算法，以上的 \{n,m} 及 * \+ 都是尽可能匹配更多次。在一些场合需求下，需要采用非贪婪算法，可用 \{-n,m} 表示尽可能匹配更少次数。\{-n,m} 也有省略变种，与 \{n,m} 用法一样，只是在左大括号内开始多一个减号。

例如，对于字符串 hello world!，如果用正则表达式 .* 或 \.\{} 就能匹配整个字符串，因为是在 标签之间贪婪匹配尽可能多的字符。但如果是 \.\{-} 则只能匹配前一个标签，即子字符串 hello，这就是非贪婪的意义。（注意：如果在 Vim 中测试该例，在 / 命令行输入这些正则表达式，须注意转义 / 本身，即应该输入 /.*<\/b>）

匹配定位点

- ^ 匹配行首
- $ 匹配行尾
- \< 匹配词首
- \> 匹配词尾

在 Vim 编辑过程中，按 * 或 # 命令，用于搜索当前光标下的单词，就会在当前单词前后自动加上 \< 与 \> 表示界定匹配整个单词。例如，你将光标移到本文的 hello 单词上，按下 *，Vim 应该会高亮所有 hello 单词，但如果有个地方写成 hello_world 加了下划线连字符，那就不会高亮这里的 hello 前缀。使用 :reg

/ 可以查看 Vim 为我们自动添加的正则表达式为 \<hello\>，你也可以按 / 进入搜索 命令后再按向上方向键把上次的搜索模式复制到当前命令行中查看。

- \zs 不匹配任何东西，只标定匹配结果的开始部分
- \ze 不匹配任何东西，只标定匹配结果的结束部分

这两个标记不影响“是否匹配”的判断，只影响若匹配成功后实际匹配的结果子字符串。例如先看个简单模式 hello.*world!，它可以匹配 hello world! 或者在这两个单词之间添加了其他乱七八糟的字符后也能匹配，匹配结果是从 hello 到 world! 之间的 所有长字符串。但是另一个类似模式\zshello\ze.*world!，它与前面那个模式能匹配一样的字符串或文本行，但是匹配结果只有前面那个 hello 单词而已。可以利用 Vim 搜索的高亮显示来理解这个差异。

所以如果仅为了搜索，\zs 与 \ze 是基本不影响结果的，但如果同时要替换时，这两个标定就很有用了，可使替换命令或函数大为简化。例如将 hello 改为首单词大写 : :s/\zshello\ze.*world!/Hello/，它只会修改后面还接了 world! 的 hello， 单独这个单词却不会被修改的。

Vim 的正则表达式，还有另外一些定位扩展，以 \% 形头的：

- \%^ 匹配文件开头
- \%$ 匹配文件结束
- \%l 匹配行，在 \% 与 l 之间应该是一个有效的数字行号，表示匹配相应的行，若在行号前再加个 < 表示匹配该行之前的行，加个 > 则表示匹配之后的行。例如 \%23l \%<23l \%>23l 等。
- \%c 匹配列，与 \%l 用法类似。
- \%# 匹配当前光标位置
- \%'m 匹配标记 m，m可以是任一个命令标记（mark）。

分组与引用

- \(\) 创建一个分组（子表达式），它本身不影响匹配，但便于其他语法功能使用
- \1 \2 … \9 依次引用前面用 \(\) 创建的分组
- \%(\) 多加一个 % 与 \(\) 创建分组一样功能，但又不当作一个子表达式，即不影响 \1 \2 等的引用次序。

分组的子表达式引用也可用于替换命令的替换部分，这可能是前向引用的常用用法，例如 :s/\(\d\+\) + \(\d\+\)/\2 + \1/ 用于将一个加法运算的表达式的两个操作数调换次序，将 123 + 321 修改为 321 + 123。

当然在正则表达内部也能用前向引用以达到某些特殊要求，比如常见的匹配 html 配对标 签，<\(.*\)>hello<\/\1> 可以匹配用任意标签括起的 hello 单词，如 hello <xyz>hello</xyz> 等，但若标签不配对不能匹配。

其他限定语法

- \c 忽略大小写
- \C 不能忽略大小写

在默认情况下，正则表达式匹配也受 &ignorecase 的影响，但如果在一个模式中任意地方加上了 \c 或 \C 控制符，就强行忽略或不忽略大小写。一般是加在表达式末尾，临时改变主意在怎么忽略大小写。

这与 \m 或 \M 的控制符不一样，它只影响后续正则表达式的魔法字符释义。不过建议放在整个正则表达式最前面为好。

Vim 正则表达式总体构成

正则表达的具体语法细节，需要经常翻手册确认。不过最后还是再归纳一下 Vim 正则表 达式的总体构成定义，按帮助文档的术语，一个正则表达式从上到下分以下几个层次：pattern <- branch <- concat <- piece <- atom <- item 。

1. 一个正则表达式也叫一个模式（pattern），一个模式可能由多个分支（branch）构成，虽然大多应用场合下只有一个分支。多个分支由 \| 分隔，表示“或”的语义（| 不是魔法字符，所以要用 \|）。任一个分支匹配目标字符串，则表示该模式匹配成功；如果多个分支都匹配，则匹配结果取第一个能匹配的分支。
2. 每个分支可能由一个或多个聚合（concat）组成，若多个聚合由 \& 分隔，这表示“ 且”的语义。必须匹配每一部分的聚合，该分支才算匹配，但匹配结果是按最后一个聚合的匹配为准。
3. 每个聚合又可以由多个分子（piece）构成，分子之间相当由自然引力结合，无须特殊字符直接粘接。如模式 abc 就只一个分支，一个聚合，该聚合有三个分子，每个分子是简单的字面字符；模式 a[0-9]c 或 a\dc同样是由三个分子组成。
4. 每个分子又由一个或多个原子（atom）构成。上一小节讲述的正则表达式语法其实主要都处于这一层。字符类别如 \d \s \w 就是表示一个原子，用 [] 指定的字符集合，也只是一个原子，而表示重复次数的 \{n,m} 就是描述多个原子的情况。
5. 每个原子即可以是普通原子，又可以是循环定义的子模式，即用 \(\) 或 \%(\) 创建的分组。

以上的第1第2层相当于逻辑或与逻辑且用于正则表达式的上层扩展，第3层却是很平凡的定义，语法细节最多的是第4层，而第5层则是更深入的高级用法。

小结

正则表达式是很精妙的技术，非短时间所能掌握，只有多加实践积累经验。在 Vim 中， 可多利用高亮模式（set hlsearch）来测试正则表达式的正确性。有其他语言或工具的正则表达式经验的用户，则特别注意一下 Vim 的特性语法。

正则表达的匹配是很复杂的算法，在其他一些语言中，可能有预编译正则表达式的功能（库函数）。但在 VimL 中，似乎还未提供类似的内建函数。不过在写较大的 VimL 脚本时 ，如果涉及使用正则表达式，也建议将常用的正则表式（字符串）统一定义在脚本开头，方便管理与修改。

第五章 VimL 函数进阶

在第二章中，我们已经讲叙了基本的函数定义与调用方法，以及一些函数属性的作用。但正如大多数编程语言一样，函数是如此普遍且重要的元素。因而本章继续讨论一些有关函数的较为高级的用法。

5.1 可变参数

可变参数的意义

一般情况下，在定义函数时指定形参，在调用函数时传入实参，且参数个数必须要与定义时指定的参数数量相等。但在一些情况下，我们将要实现的函数功能，它的参数个数可能是不确定，或者有些参数是可选可缺省使用默认值。这时，在函数定义中引入可变参数就非常方便了。相对于可变参数，常规的形参也就命名参数。

- 在函数头中，用三个点号 `...` 表示可变参数，可变参数必须用于最后一个形参，如果有其他命名参数，则必须位于 `...` 之前。
- 在函数体中，分别用 `a:1` `a:2` …… 等表示第一个、第二个可变参数。用 `a:0` 表 示可变参数的数量，`a:000` 是由所有可变参数组成的列表变量。
- 命名参数最多允许 20 个，虽然大部分情况也够用了。可变参数的数量没有明确限制。
- 调用函数时，传入的实参数量至少不低于命名参数的数量，但传入的可变参数数量可以为 0 或多个。当没有传入可变参数时，`a:0` 的值为 0。

需要强调的是，只有定义了 `...` 可变参数，才能在函数体中使用 `a:0` `a:000` `a:1` 等特殊变量。较好的实践是先用 `a:0` 判断可变参数个数，然后视情况使用 `a:1` `a:2` 等每个可变参数。如果只传入一个实参，与使用于 `a:2` 变量，会发生运行时错误。此外 `a:000` 就当作普通列表变量使用好了，`a:000[0]` 就是 `a:1`，因为列表元素索引从 0 开始。

例如，可用以下函数展示可变参数的使用方法：

```
function! UseVarargin(named, ...)
    echo 'named argin: ' . string(a:named)

    if a:0 >= 1
        echo 'first varargin: ' . string(a:1)
    endif
    if a:0 >= 2
        echo 'second varargin: ' . string(a:2)
    endif

    echo 'have varargin: ' . a:0
    for l:arg in a:000
        echo 'iterate varargin: ' . string(l:arg)
    endfor
endfunction
```

你可以用 `:call` 调用这个函数，尝试传入不同的参数，观察其输出。可见有两种写法获取某个可变参数，比如用 `a:1` 或 `a:000[0]`，视业务具体情况用哪种更方便。而且 `a:000` 还可用列表迭代方法获取每个可变参数。

不定参数示例

在 2.4 节，我们已经定义了一个演示之用的函数 `Sum` 可计算两个数之和，简化重新截录于下：

```
function! Sum(x, y)
    let l:sum = a:x + a:y
    return l:sum
endfunction
```

现假设要计算任意个数之和，则可改为如下定义：

```
function! Sum(x, y, ...)
    let l:sum = a:x + a:y
    for l:arg in a:000
        let l:sum += l:arg
    endfor
    return l:sum
endfunction
```

这里认为调用 Sum() 时必须提供两个参数，否则求和没有意义。其实也可以定义为 Sum(...)，将函数实现中的 l:sum 初始化为 0 即可。

若一个函数用 Fun(...) 定义，只声明了可变参数，则可用任意个参数调用，非常通用。然而过于通用也表明意义不明确，良好的实践是，除非有必要，尽可能用命名参数，少用可变参数。使用合适的参数变量名，函数的可读性增强，使用可变参数时，最好加以注释；同时也建议在函数前面部分判断可变参数数量与数量，第一时间分别赋于另外的局部变量，也能增加函数的可读性。

调用这个求和函数时，用 :call Sum(1, 2, 3, 4) 方式。事实上，只为这个需求的话，不必用可变参数，直接用一个列表变量作为参数可能更方便。如改写为：

```
function! SumA(args)
    let l:sum = 0
    for l:arg in a:args
        let l:sum += l:arg
    endfor
    return l:sum
endfunction
```

这个函数的意义是为一个列表变量内所有元素求和，以 :call Sum([1, 2, 3, 4]) 方式调用。

然而需要注意的是，并非所有用可变参数的函数，都适合将可变参数改为一个列表变量。

默认参数示例

在 VimL 的内置函数中，格式化字符串的 printf() 就是接收任意个参数的例子。另外还有大量内置函数是支持默认参数的，如将列表所有元素连接成一个字符串的 join()。这种情况与不定参数略有不同，它能接收的有效参数个数是确实的，只是在调用时后面一个或几个参数可以省略不传，不传实参的话就自动采用了某个默认值而已。

比如我们也可以自己实现一个类似的函数 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
```

虽然可以（更低效率）用循环连接字符串，但这时为简明说明问题，直接调用内置的 join() 完成实际工作了。关键点是提供了另一个逗号作为默认分隔字符，通过 a:0 来判断传入的可变参数个数，再给分隔字符赋以合适的初始值。 其实这个 if 分支可以直接用 get() 函数代替：let l:sep = get(a:000, 0, ',')。这用起来更为 简洁，不过用 if 分支明确写出来，更容易扩充其他逻辑，即使是用 echo 打印个简 单的日志。

间接调用含可变参数的函数

一般情况下，函数都不是独立完成工作的，往往还需要调用其他的函数。假如一个支持可变参数的函数内，要调用另一个支持可变参数的函数，给后者传递的参数依赖于前者接收的不确定的参数，这情况就似乎变得复杂了。

为说明这种应用场景，先参照上述 Sum() 函数再定义一个类似的连乘函数：

```
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，却不支持 *= 操作符，请参阅 :h += 。

然后再定义一个更上层的函数，根据一个参数分发调用连加 Sum() 或 连乘 Prod() 函数，传入剩余的不定参数：

```
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)
```

在这个示例函数中，第一行的 echo 语句用于调试打印，不论是用刚才自定义的 Join() 或内置的 join() 函数都能正常工作。但是在随后的 if 分支中， 不论是 Sum(...) 还是 Sum(a:000) 都不能达到预期效果，虽然它作为“伪代码”很好地表达了使用意途，所以先将其注释了。

先分析原因，Sum(...) 是语法错误。因为 ... 只能用于函数头表示不定参数，却不能在函数体中表示接收的所有不定参数。a:000 可以表示所有不定参数，但它只是一个列表变量，调用 Sum(a:0000) 时只传了一个参数变量，而原来定义的 Sum() 函数要 求至少两个参数，所以也会出错误，因为相当于调用 Sum([1,2,3,4]) 也是错误的。

解决办法是用 call() 函数间接调用，它的第一个参数是一个函数，第二个参数正是一个列表，这个列表内的所有元素将传入第一个参数所代表的函数进行调用。例如，这语句 :echo call('Sum', [1,2,3,4]) 能正常工作。于是可将 Calculate() 函数改写：

```
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
```

这里再作了另一个优化，先对不定参数个数作了判断，不足 2 个时返回错误。

第五章 VimL 函数进阶

5.2 函数引用

关于函数引用的帮助文档先给传送门 :h FuncRef（注意大写）。很多函数的高级用法都在函数引用基础上建立的。

函数引用的意义

继续接着上一节的内容引申来讲。例如在 Calculate() 函数中间接调用 Sum() 时须 用如下语法： call('Sum', a:000) ， 'Sum' 函数名须用引号括起来当作一个字符串参数传入。

如果尝试执行 :echo call(Sum, [1,2,3,4]) 就会报 E121 的“未定义变量”错误。也

就是说，Sum 是一个自己定义的函数，但函数与变量在 VimL 中有本质的不同，而 call() 要求一个变量作为参数，所以不能直接将函数传入。然而这个变量又要求能代表函数，所以 VimL 就需要一个“函数引用”的概念。

就这个特殊的 call() 而言，在第一参数中将一个函数名用引号括起的字符串也能达到引用一个函数的目的，但这显然是不正式不通用的。函数引用也是一个变量，不过是另一种特殊的变量（值）类型，不应该与简单的字符串变量类型混淆。

在其他一些编程（脚本）语言中，函数是所谓的一等公民，即与变量的地位一样，可以用变量的地方，也可以用函数。但在 VimL 设计之初，函数与变量两个不同次元的东西。只有在引入了函数引用之后，函数引用与变量才是相同的东西。

函数引用的定义

可以内置函数 function() 创建一个函数引用，其参数就是所要引用的函数名（引号字符串），即可以是内置函数也可以是自定义函数的名字。例如：

```
: let Fnr_Sum = function('Sum')
: echo type(Fnr_Sum)
: echo Fnr_Sum(1,2,3,4)
```

上例创建一个变量 Fnr_Sum，它引用自定义函数 Sum()。查看这个变量的类型，显示是 2，这就是函数引用的类型（v:t_func）。然后这个函数引用可以像原来那个一样调用，也就是后面接括号传入参数列表。

函数引用变量与函数本身的关系，就与之前所述的列表（或字典）变量与列表（或字典）实体之间的关系。在常规运用场合中，一般可不必理会其中的差异，凡是要求函数调用的地方，都可以用函数引用代替。而且，函数引用作为一个变量，使用范围将更加灵活。因为 VimL 的变量是弱类型的，在使用变量时不检查变量类型，所以在任何使用变量的地方，也都可以使用函数引用代替。当然，你不能试图对函数使用进行加减乘除这样的操作，那会触发运行时错误，函数引用主要（也许是唯一）支持的操作就是调用。

VimL 的变量名自有其规则（见第二章），而函数引用的变量名在此规则上还有更严格一点的限制，就是也必须也以大写字母开头。这是因为要与函数名的规则吻合。因为从代码语法上看一个函数调用，无从分辨它是函数引用还是函数本身。主要注意如下几点：

- 函数引用变量也可以加作用域前缀，如果加了 s: w: t: 或 t: 这几个前缀， 则不再要求变量名主体以大写字母开始了，因为这种情况下不会有歧义。参数作用域前缀 a: 用于函数引用之前，也不必大写字母。
- 如果在函数引用变量名之前加全局作用域前缀 g: 或局部作用域前缀 l:，仍然要 求其变量名主体以大写字母开头。因为这两种前缀是可以省略的，要保证省略后的等价的“裸”调用仍然合乎函数调用规则。
- 函数引用变量名，不能与已有的自定义函数名相同，否则也会发生歧义，vim 将无从分 辨是触发调用函数引用呢，还是触发调用同名函数本身。
- 函数引用变量名允许与已存在的其他变量名重名，只不过其含义是重定义或覆盖原变量的意义，虽然语法上合法，但不建议这么做。

再次提醒一下，function 这个“关键字”，即是一个命令名，也是一个内置函数名。用 :function 命令是创建或定义一个函数，则 function() 函数则是创建或定义一个函数引用（其参数须是已由 :function 命令创建的函数名，或内置函数名）。命令与函

数是完全不同空间次元的东西，也与变量互不相关。如果你愿意，甚至也可以自定义一个叫 function 的变量，但最好不要这样做。

在 VimL 中，有很多内置函数与命令重名，用于实现相似的功能。上节刚用到过的 call() 函数与 :call 命令也是这种情况。在查 vim 帮助文档时，查函数时在后面加对空括号，查命令时在前面加个冒号。另一方面，VimL 的内置变量名都是以 v: 前 缀的，这倒不必担心混淆。

函数引用的使用

下面再讲解函数引用的使用建议与示例。仍以上节末用于实现不定参数连加或连乘的 Calculate() 函数为例。

将函数引用作为参数传递

首先，不建议使用全局的函数引用变量。因为用 :function 命令定义的函数是全局的，尽量不要将函数引用也定义在全局作用域中，避免麻烦。例如，可将上节的 Calculate() 函数改为如下使用函数引用的方式（为简便起见，略过参数检测）：

```
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
```

这里先根据参数一创建一个函数引用 Fnr，在函数内定义的变量都是局部变量，l: 前缀可选。然后这个函数引用也可以作为参数传给 call() 函数，它能同时处理作为函数名的字符串变量类型或函数引用类型，反正都是用以访问实际所调函数的手段；也不妨认为在之前传入字符串时，call() 函数也会自动先调用 function() 获得函数引用。

脚本局部函数及引用

上面改写的 CalculateR() 函数有一处不太好，就是每次调用都要重新创建 l:Fnr 这个相同的函数引用变量，略显低效。在实践中，函数定义一般是写在单独的脚本中，因此函数引用也可以定义为 s: 脚本局部变量。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 ==# '*'
        let l:Fnr = s:fnrProd
    endif

    let l:result = call(l:Fnr, a:000)
    return l:result
endfunction
```

注意，如前所述，s: 前缀的函数引用变量可用小写开头，l: 或缺省前缀的函数引用须大写开头。这里主要为演示不同前缀的函数函数引用变量，其实 l:Fnr 中间变量也 可省去，直接将 call() 调用语用写在 if 分支中。

这样，s:fnrSum 与 s:fnrProd 函数（引用）就是私有的了，只能在该脚本内使用，而 CalculateRs() 函数仍定义为全局函数，提供为外部公用接口。但是，那两个私有变量引用的仍是公用的函数 Sum() 与 Prod()。如果想再要隐藏，可以将这两个函数也定义为 s: 的作用域：

```
" File: ~/.vim/vimllearn/funcref.vim

function! s:sum(...)
    let l:sum = 0
    for l:arg in a:000
        let l:sum += l:arg
    endfor
    return l:sum
endfunction

function! s:prod(...)
    let l:prod = 1
    for l:arg in a:000
        let l:prod = l:prod * l:arg
    endfor
    return l:prod
endfunction

let s:fnrSum = function('s:sum')
let s:fnrProd = function('s:prod')

echo s:
```

这里的 s:sum() 函数对原 Sum() 略有修改，不再强制要求至少两个参数。同时函数名加上 s: 前缀后，也不再强制要求以大定字母开头。当用 function() 创建函数引 用时，须将 's:sum' 整个字符串当作该脚本局部数字的“名字”传入为参数。

然后，重点迷惑来了，脚本内的 s:sum() 实际函数名其实并不是 's:sum'! 这只是语法上规定的书写文法。在 vim 内部，会将 s: 前缀的函数名替换为 <SNR> _。其中编号是指 vim 在加载该文件时对其赋与的编号。可用 :scriptnames

命令查看当 前 vim 所加载过的所有脚本，一般情况下编号为 1 的第一个加载文件就是你的起始配置文件 vimrc，然后每次加载脚本时顺序编号。所以 s:sum() 脚本私有函数的实际名字是动态变化的，在不同的 vim 会话中加载时机极可能不一样，其编号中缀也就不一样了。

如果在脚本末尾加上 echo s: 这个语句（s: 是一个特殊字典，保存着该脚本内定义的所有以 s: 前缀开始的脚本局部变量），那么在加载该脚本时，将回显如下信息：

```
{'fnrSum': function('<SNR>77_sum'), 'fnrProd': function('<SNR>77_prod')}
```

表明在这次 vim 会话环境中，s:sum() 函数名实际上是 <SNR>77_sum，也可以直接用这个名字来调用该函数，如在命令行中输入

```
: echo <SNR>77_sum(1, 2, 3, 4)
```

是能正常工作中的。

因此，看似脚本局部私有的 s:sum() 实际上是被转化成了 <SNR>77_sum() 全局公有 函数。其中 <SNR>77_ 前缀在在某些地方也可用特殊符号 <SID> 表示。当然，任何 正常的人，都不会采用后者来调用函数，况且脚本编号都是临时赋与的不保存一致性，于是也算达到了作用域隐藏的目的。

另外，还有一点要注意的是，s:sum() 是函数，不是变量，所以它不会被保存在 s: 字典内。只有函数引用变量 s:fnrSum 与 s:fnrProd 才保存在 s: 字典内，其键 就是变量名 fnrSum 与 fnrProd，其值就是相应的函数引用。显然，vim 不能自作主 张地自动为 s:sum() 创建一个名为 s:sum 的函数引用变量，甚至我们自己也不能手动用 :let ... function() 语句创建名为 s:sum 的函数引用变量，否则在调用 s:sum(1,2,3,4) 是就会发生语法歧义。但是，我们能用它创建其他类型的变量，如在脚本末尾加入如下代码并重新用 :source 加载：

```
" File: ~/.vim/vimllearn/funcref.vim

" let s:sum = function('s:sum') "
" let s:prod = function('s:prod')

let s:sum = '1+2+3+4'
let s:prod = '1*2*3*4'
echo s:

echo s:sum(1,2,3,4)
echo s:prod(1,2,3,4)
```

可以把 s:sum 赋值为字符串类型变量，然后 s:sum() 函数并未失去定义，仍然可正常调用。所以，s: 作用域前缀用于变量与函数前有着不同的实现意义。s:sum() 函 数本质上是 <SNR>77_sum() 函数，与 s:sum 变量大有不同。然而，正常的程序猿非常不建议玩这样的杂耍。

将函数引用收集在列表中

在前一示例中，在脚本中创建的 s: 前缀的函数引用变量，被自动地收集保存在一个特

殊字典中。这表明函数引用与普通变量“无差别”的同等地位，可以用在任何需要变量的地方。比如，我们也可以主动地将函数引用保存在一个列表中，以实现某些特殊功能：

```
" File: ~/.vim/vimllearn/funcref.vim

let s:operator = [function('s:sum'), function('s:prod')]
function! CalculateA(...)
    for l:Operator in s:operator
        let l:result = call(l:Operator, a:000)
        echo l:result
    endfor
endfunction
```

这里，我们定义了一个列表变量 s:operator ，其元素都是能接收不定参数的运算函数的引用。然后在函数 CalculateA() 中遍历该列表，为每个函数传递参数进行计算。这是个全局函数，所以加载脚本后，可直接在命令行中执行 :call CalculateA(1,2,3,4) 验看结果。

仍然要注意的是，在 for 循环中，循环变量 l:Operater 仍然要以大写字母开头，才能接收 s:operator 列表内的函数引用变量。否则，若以小写字母的话，有可能省去 l: 前缀，写出类似 operator(1,2,3,4) 的函数调用，这就有语法错误了，因为小写字母的函数名调用，都保留给 VimL 的内置函数。

良好的实践是，始终以大写字母开头命名函数引用变量，不管什么作用域前缀；如果不嫌麻烦，再以 Fnr 为变量名前缀也未尝不可。

第五章 VimL 函数进阶

5.3 字典函数

函数引用能保存在字典，这不意外，上节就提到过，脚本内定义的 s: 前缀变量（包括函数引用），就自动保存在 s: 这个特殊字典中。关键是如何主动利用这个特性，为编程需求带来便利。在本节中，将保存在字典键中的函数引用简称为字典函数。

将已有函数保存在字典中

沿用上节的示例，将函数引用保存在字典中，相关代码改写如下：

```
" >>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
```

这里先定义了一个字典变量 s:dOperator，并用键 + 保存函数 s:sum() 的引用，用键 * 保存函数 s:prod() 的引用。然后改写 CalculateD() 函数就很简洁了，根据传入的第一参数索引字典，获得相应的函数引用，再调用之。因为直接用键索引字典，且认为没有遍历全部键的需求，所以还可以在 s:dOperator 字典加入非函数引用的键，比如 desc 保存了一条描述，字符串类型。

可以在命令行中输入 :echo CalculateD('*', 1, 2, 3, 4) 测验一下。注意到该函数没有检查传入参数是否有效的键，如 :echo CalculateD('**', 1, 2, 3, 4) 会报错。可以先用 has_key() 内置函数检查参数 a:operator 是否存在的键，更进一步，可再用 type() 函数与该键相关联的值是否函数引用。如果参数是非法的，则提前返回，至少返回什么值表示错误，那就与具体需求有关了。也许在某些情况下，不检查参数，直接让它在出错时终止脚本运行也是可接受的处理方式。

按成员的方式引用函数

我们知道，字典元素有两种索引方式，一是用方括号（类似列表索引），一种是用点号（类似成员索引）。不过后者只是前者的语法糖，要求键名是简单字符串（有效标志符）。因此可以用一个较有意义单词键名来代替 + * 符号键名，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)
```

如果之前没有在字典中定义 + 键，也可以直接用 let s:dOperator.sumFnr = function('s:sum') 获得函数引用。这里以小写字母开头的键名也可以保存函数引用。然后调用函数的写法就是 s:dOperator.sumFnr()。由于使用的是脚本局部的字典变量，须用 :source 命令重新加载脚本文件执行上例，观察这种调用方法的结果。

直接定义字典函数

为了在字典键中保存一个函数引用，之前其实成分了三步工作：

1. 用 :function 命令定义一个函数；
2. 用 function() 函数获取这个函数的引用；
3. 用 :let 命令将这个函数引用赋值给字典的某个键。

但这三步曲（实际是两条语句）可以合起来，直接在定义函数时就将其引用保存在字典中，其语法示例如下：

```
" >>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)
```

其实就相当于将之前的函数头 `:function s:sum(...)` 改为 `:function s:dOperator.sum(...)`，函数体功能实现完全一样。要注意的是在执行这一行之前，`s:dOprator` 字典必须是已定义的。然后调用该函数的用法完全一样。

请注意区分一下，`s:dOperator.sumFnr` 显然是一个函数引用，它引用事先已定义的 `s:sum()` 函数。`s:dOperator.sum` 也是一个函数引用，它引用的又是哪个函数呢？它引用的是即时定义的函数，它没有名字（没机会也没必要给个名字），也叫做匿名函数。在 Vim 内部，它将这样定义的匿名函数一个编号，所以也叫编号函数。

如果在脚本文件末尾写上 `echo s:` 这条语句，根据其输出结果，就能更清楚地分辨这些函数引用变量的异同。例如，执行结果大概相当于如下定义：

```
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['+']` 与 `s:dOperator.sumFnr` 都是引用同一个函数，那就是 `s:sum()` 局部函数，不过 vim 自动将其修正为 `<SNR>77_sum()` 全局函数 。而 `s:dOperator.sum` 则完全引用另一个函数，是编号为 `172` 的匿名函数。当然， 你的输出中，脚本编号与函数编号极可能是不一样的。

我们知道，退化的 `:function` 命令可以查看打印函数定义。所以可以用 `:function <SNR>77_sum` 在命令行直接执行，其输出应该与脚本中定义的 `s:sum()` 函数一致。但 是在命令行使用 `:function s:sum` 是错误。那匿名函数怎么查看呢，直接用编号作为参数是不行的，需用一个大括号括起来，如：

```
: function <SNR>77_sum
: function {173}
```

但是，用于获取一个函数引用的 function() 却无有效方法仅从匿名函数的编号获得其引用。如 function('173') 或 function('{173}') 都不能正常工作。匿名函数一般必须在创建时赋值给某个函数引用变量，然后只能通过该函数引用调用之。当然了，该函数引用可以再赋值给其他变量就是。

字典函数的特殊属性

如果仔细观察上述 :function {173} 命令输出，可以发现它在函数头定义行尾，自己添加了一个关键字 dict，表示将要定义的函数具有 dict 属性。这个属性指出该函数必须通过字典来激活调用，也就是说必须将其引用保存在字典的某个键中。然后在函数体中，可以使用 self 这个关键字，它表示调用该函数时所用到的字典变量。

例如，假设我们要在上述 s:dOperator 字典中另外加一个计算圆面积的函数。从数学上讲，圆面积只是其半径的函数，应该只要传入半径参数。但在程序中实现计算时，还要涉及一个圆周率常量。这个常量不适合放在函数内定义，当然可以定义为 s: 脚本变量 ，不过最好还是保存在同一个字典中。

```
" >>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)
```

我们先定义了 s:dOperator.area 函数（引用），然后调用 s:dOperator.area(2) 来 计算半径为 2 的圆面积。在函数定义体内用到了 self.PI，这个 self 就是调用 该函数时所用到的字典变量，也即 s:dOperator。

这里，我们调用时与定义时用到的字典变量是同一个，但这不是必须的。比如，我们可以创建另一个字典 s:Math，它保存了一个 PI 键，为示区别，这个 PI 保存的圆周率精度大一些：

```
" >>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)
```

请观察 s:dOperator.area(2) 与 s:Math.Area(2) 计算结果的不同，表明后者调用时 self.PI 确实用到了 s:Math.PI 的值，而不是 s:dOperator.PI 的值。而且， 在 s:Math 中的函数名 Area 不一定要与最初定义时所用的 area 相同。但是函数 体内用到的 PI 键名，必须相同。

如果把 s:dOperator.area 这个函数（引用）赋值给普通变量（非字典键），会发生什么情况呢？尝试在脚本末尾继续添加如下代码并加载运行：

```
let g:Fnr = s:dOperator.area
echo g:Fnr(2)
```

结果它会报 E725 错误，提出不能在没有字典的情况下调用具有 dict 属性的函数。这似乎很好理解，因为在 area() 函数体内，用到了 self.PI，没有字典的话，这个 self 就无所引用了。实际上，即使在函数体内没有到用 self ，也不能绕过字典去调用字典函数。比如原来的 s:dOperator.sum() 就没用到 self，但如下代码也时非法的：

```
let g:Fnr = s:dOperator.sum
echo g:Fnr(1,2,3,4)
```

在为 g:Fnr 赋值时不会出错，在调用 g:Fnr() 时才出错。所以 vim 是通过 dict 这个函数属性来检测调用合法性的，因为这种函数体内有可能用到 self，提前终止潜在的错误，总是更安全的设计。而且，既然用到 dict ，就意味着大概率会用到 self， 否则将一个非 dict 属性的函数保存在字典中，是很无趣的（虽然合法）。以下语句却不会出错：

```
let g:Fnr = s:dOperator.sumFnr
echo g:Fnr(1,2,3,4)
```

因为 s:dOperator.sumFnr 所引用的函数其实是 s:sum()，它在定义时未指定 dict 属性。所以 s:dOperator.sunFnr 只起来到一个传递变量值的中介作用，g:Fnr 也是 对 s:sum() 的函数引用，当然也就可以直接调用了。

普通函数的字典属性

上面在定义 s:dOperator.sum 与 s:dOperator.area （对匿名函数的引用）时，并未显式写出 dict 属性。这只是 :function 定义字典函数时的语法糖，vim 会自动 添加 dict 属性。

定义普通函数时也可以指定 dict 属性，例如我们另外写个计算矩形面积的函数：

```
function! s:area(width, height) dict
    return a:width * a:height
endfunction

" echo s:area(3, 4) |"

let s:Rect = {}
let s:Rect.area = function('s:area')
echo s:Rect.area(3, 4) |"
```

但是，由于 s:area() 函数是 dict 属性的，所以直接调用 s:area() 会出误。必 须把它（的引用）放在一个字典中，如上为此专门建了个空字典变量 s:Rect，将函数 引用保存在其 area 键名中，才能调用 s:Rect.area()。

因此，当一个普通函数用了 dict 属性，却没用到 self 特性，好像用处不是很大，反而限制了其正常使用。为此，将 s:area() 函数重新定义如下：

```
function! s:area() dict
    return self.width * self.height
endfunction

let s:Rect.width = 3
let s:Rect.height = 4
echo s:Rect.area()
```

取消 s:area() 的函数参数，而将 width 与 height 参数保存在 s:Rect 字典中，然后就可以无参调用 s:Rect.area() 了。这样，长、宽就相当于矩形（s:Rect）的属性，而求面积的 area() 就相当于它的方法。这就初具面向对象的特征了（这将在后续章节中再详细讨论）。

注意这里的 s:area() 函数体内用到了 self，则在函数头一定要指定 dict 属性。反之则不强制要求。

具有 dict 属性的函数，除了对过字典键引用来调用外，也可以用 call() 函数间接 调用。之前已经介绍过 call() 函数，其实它还可接收第三个可选参数，按 :help call() 介绍其用法是 call({func}, {arglist} [, {dict}])。如果第一个参数（函数名或函数引用）所指代的函数具有 dict 属性，第三个参数就应该提供一个字典传递给这个函数体实现中的 self 变量。

因此，第二个版本（无参数）的 s:area() 可以这么调用：

```
echo call('s:area', [], s:Rect)
echo call(function('s:area'), [], s:Rect)
```

这两条语句都合法，不过由于使用了 s:area 字符串，必须在脚本中才能运行。当 call() 在调用 s:area() 时，s:area() 函数内的 self 也就是 s:Rect 了。

至于第一个版本带两个参数的 s:area() 则可以这么调用：

```
echo call('s:area', [5, 6], {})
echo call('s:area', [5, 6]) |"
```

将参数收集在一个列表变量中，作为第二参数传入。由于函数体内未用到 self，在第 三参数随便提供一个字典变量就行，即使是个空字典 {}。但若不提供这个字典参数，则会发生运行时错误。

直接定义字典函数与间接定义的比较

综上再小结一下，定义字典函数（引用）有两种方式。一是直接用一条语句搞定，字典键引用了一个匿名函数；二是先定义函数，再将该有名函数的引用赋值给字典键。不妨分别称之为直接定义与间接定义。

- 直接定义：function dict.method()
- 间接定义：function Method() 与 let dict.method = function('Method')

显然，直接定义的语法更简洁方便，请尽量使用这种语法。那么间接定义的写法还有没有什么存在的意义呢？

首先，这可能是历史原因。VimL 也是随 Vim 逐步发展完善起来的，很有可能函数引用的概念先于 dict 属性与 self 变量的引入。因而也就先有分步写的字典函数引用，然后才有一步到位的语法糖写法。

其次，间接定义的函数引用有更灵活的控制权。直接定义的字典函数必定是匿名函数的引用，且隐含具有 dict 的属性，不论是否显式写出该关键词。这也就意味着不能将直接定义的字典函数引用赋值给普通函数引用变量，那是不能工作的。但在间接定义字典函数时有更多的选择，在定义函数时可根据需要是否指定 dict 属性。没有 dict 属性的 函数引用可以赋值给普通变量。因此，从编码实践上建议：

- 直接定义的字典函数，也始终显式加上 dict 关键词，不要太依赖语言的隐式作用。
- 普通函数，如果实现体中需要用到 self 才加 dict 属性关键词。

最后，字典键名引用有名或匿名函数，会影响调试与错误信息。通过示例详细说明，将以下代码片断添加到本节的演示脚本末尾，并用 :source 重新加载。

```
" >>File: ~/.vim/vimllearn/funcref.vim

function! s:Rect.debug1() dict abort
    echo expand('<sfile>')
    Hello Vim,
endfunction

function! s:debug2() abort
    echo expand('<sfile>')
    Hello Vim,
endfunction
let s:Rect.debug2 = function('s:debug2')

function! s:Rect.test() dict " abort
    echo expand('<sfile>')
    call self.debug1()
    call self.debug2()
endfunction

function! s:test() abort
    echo expand('<sfile>')
    call s:Rect.test()
endfunction

function! Test() abort
    echo expand('<sfile>')
    call s:test()
endfunction
```

```
echo expand('<sfile>')
```

复用原来的字典 s:Rect，增加了两个函数引用键，其中 debug1 是直接定义的，debug2 是间接引用 s:debug2() 的。这两个函数内随意加了一行错误语句。这在加载脚本时并不会出错误，只有实际调用了相应函数才有机会出错。然后再定义了一个统一的 s:Rect.test() 函数，在其内调用这两个 debug 函数。最后还定义了 s:test() 与 Test() 函数。只有 Test() 是全局的，可以在命令行中执行 :call Test() 查 看结果。在执行前先人工分析下这将发生的函数调用链：

```
   Test() -->    s:text() -->    s:Rect.test()
[1] -->    s:Rect.debug1() |
[2] -->    s:Rect.debug2() |   s:debug2()
```

我这里执行 :call Test() 后输出如下，脚本编号与函数编号肯定是依环境不同的：

```
function Test
function Test[2]..<SNR>77_test
function Test[2]..<SNR>77_test[2]..181
function Test[2]..<SNR>77_test[2]..181[2]..180
Error detected while processing function Test[2]..<SNR>77_test[2]..181[2]..180:
line    2:
E492: Not an editor command:     Hello Vim,
function Test[2]..<SNR>77_test[2]..181[3]..<SNR>77_debug2
Error detected while processing function Test[2]..<SNR>77_test[2]..181[3]..<SNR>77_debug2:
line    2:
E492: Not an editor command:     Hello Vim,
```

其中，常规字体是各函数内 echo expand('<sfile>') 的正常输出，红字部分是错误语句触发的输出，即 vim 自动给出的错误提示信息。主要是触发 E492 这个错误，它说 Helle Vim 不是编辑器的有效命令。并在之前先打印出错时所在的函数名与行号。重点关注一下函数名的表示方法，例如在 s:Rect.debug1() 出错时的位置信息：

```
function Test[2]..<SNR>77_test[2]..181[2]..180:
```

对比之前的分析，第一层调用是全局函数 Test，中括号 [2] 表示在第二行调用下一层函数，即 s:test()，它被转化成 <SNR>77_test 函数名，然后第二行再调用 s:Rect.test() ，这是匿名函数，所以只能打印出编号 181，然后继续调用 s:Rect.debug1() ，它也是匿名函数，也只打印出编号 180。到这个函数就出错了，没能再调用其他函数，出错行号另起一行打印出来。

在 s:Rect.debug2() 出错时的位置信息类似：

```
function Test[2]..<SNR>77_test[2]..181[3]..<SNR>77_debug2:
```

只不过在倒数第二层的行号从第二行改为了第三行，最后一个函数名打印出了实际所引用的函数名 <SNR>77_debug2 ，也就是脚本中的 s:debug()。

这有什么差别呢？试想我们若用 VimL 开发实用功能（主要是插件时），调用链经常也会这么长或者更长。当 vim 报错时，给出一长串错误提示，我们第一反应是想知道哪里出错了，最终出错在哪个函数中。这反映在出错信息的最后一个调用函数，但是像 s:Rect.debug1() 这样的直接定义的字典函数，vim 只打印个 180 编号，可能完全

不知所云。而像 s:Rect.debug2() 这个间接定义的字典函数，它会打印出函数名。即使你也不知脚本编号，那也是有迹可循，比如用 :scriptnames 检查。而且在实践中，你也不可能在很多不同脚本中都定义了相当的函数，那么不用检查脚本编号也基本能定位错误了。

还有重要一点，在开发 VimL 脚本过程中，如果修改 Bug 后重新加载脚本，那直接定义的字典函数所引用的匿名函数编号是会变化的。因为它相当于重新定义了另一个匿名函数并为字典键赋值，而原来那个匿名函数再无引用无可访问就会自动释放（垃圾回收机制）。但是，脚本编号并不会改变，除非大重构把文件名也改了。这种编号的变化性对查 Bug 也多少会有影响的。

顺便提一下，也许你也注意到了，vim 自动打印的出错位置信息，其实就是 <sfile> 的值。如果用在函数中，那就是运行到该处时完整的调用链字符串；在不同时刻从不同入口调用时还可能给出不同的值。但如果用在函数外，那就只能是在脚本文件中，<sfile> 就表示脚本文件名（故不能直接用在命令行中）。这也是 sfile 这个单词意义的来源。不过你也可以将脚本整体理解为一个函数（也是一个执行单元），其“函数名”显然就是脚本名了。

还有一点得注意，在定义 s:Rect.test() 函数时，没有加 abort 属性。按之前的建议，定义函数时始终加 abort 是良好的习惯，因为它会在出错时立即终止运行，避免更多的错乱。不过在这里，如果有 abort 属性，它在调用 self.debug1() 出错后就 立即终止，self.debug2() 也就没机会调用了。由于我们想对比出错信息，要求触发所有错误，因而特意取消 abort 属性。

第五章 VimL 函数进阶

5.4* 闭包函数

自 Vim8，进一步扩展与完善了函数引用的概念，并增加了对闭包与 lambda 表达式的支 持。请用 :version 命令确认编译版本有 +lambda 特性支持。

闭包函数定义

学习 Vim 新功能，在线帮助文档是最佳资料。查阅 Vim8 的 :help :function，可发现在定义函数时，除了原有的几个属性 range abort dict 外，还多了一个 closure 属性。这就是定义闭包函数的关键字。并给出了一个示例，我们先将其复制到一个脚本中并执行：

```
" >File: ~/.vim/vimllearn/closure.vim

function! Foo()
    let x = 0
    function! Bar() closure
        let x += 1
        return x
    endfunction
    return funcref('Bar')
endfunction
```

这里有几点需要说明：

- 函数可以嵌套了，在一个函数体内可以再定义另一个函数。
- 内层函数 Bar() 指定了 closure 属性，就是将其定义为闭包函数。
- 在内层闭包函数 Bar() 中，可以使用外层环境函数 Foo() 的局部变量 x。
- 外层函数返回的是内层函数的引用。
- 当 Foo() 函数返回后，在 Bar() 内仍然可正常使用局部变量 x。

现在来使用这个闭包，可在命令行中直接输入以下语句试运行：

```
let Fn = Foo()
echo Fn()
echo Fn()
echo Fn()
```

可见，在每次调用 Fn()，也就是调用 Bar() 时，它会返回递增的自然数，在两次调用之间，会记住变量 x 的值。对比普通函数，当其返回后，其部分变量就离开作用域不再可见，每次调用必须重新创建与初始化局部变量。而 Bar() 函数能记住 x 变量 的状态，就是由于 closure 关键字的作用。

除些之外，Bar() 就与普通函数一样了。特别地，它的函数全名就是 'Bar'，即它也是个全局函数，也可以直接在命令行调用。如下语句依然正常地输出递增自然数：

```
echo Bar()
echo Bar()
echo Fn()
```

另外必须指出的是，在 Foo() 函数内创建 Bar() 引用时，用的是 funcref() 函数，而不是 function() 函数。funcref() 也是 Vim8 才引入的内置函数，它与之前的 function() 函数功能一样，也就是创建一个函数引用。只有一个差别，function() 只简单地按函数名寻找它所“引用”的函数，而 funcref() 是按真正的函 数引用寻找目标函数。这其中的差别只在原函数被重定义了才能体现。

例如，我们再用 function() 创建一个类似的闭包函数引用，为示区别每次递增 2。将 以下代码附加在原脚本之后，再次加载运行。

```
" >>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()
```

初步看来，Goo() 函数能与 Foo() 完全一样地使用，获取一个闭包引用，依次调用，并且可与所引函数 Bar() 交替调用，也能保持正确的状态。

但要注意，在 Goo() 函数内定义的闭包函数也是 Bar() 。所以在每次调用 Goo() 或 Foo() 都会重新定义全局函数 Bar()。如果用 function() 获取 Bar() 的引 用，它就是使用最新的函数定义。如果用 funcref() 获取 Bar() 的引用，它就一直 使用当时的函数定义。

例如，我们直接在外面再次重定义一下 Bar() 函数：

```
function! Bar()
    return 'Bar() redefined'
endfunction

echo Bar()
echo Fn()
echo Gn()
```

运行结果表明，Fn() 能继续递增数值，但 Gn() 却调用了重新定义的函数，失去了递增的原意。

所以，为了保证闭包函数的稳定性，务必使用新函数 funcref() ，而不要用旧函数 function()。当然，function() 函数除了为保证兼容性外，应该也还有其适合场景。

另外，非常不建议直接调用闭包函数，应该坚持只通过函数引用变量来调用闭包。但是，目前的 VimL 语法，似乎没法完全阻止直接调用闭包。因为 :function 定义的是函数，而非变量，不能为函数名添加 l: 前缀来限制其作用域。可以加 s: 定义为脚本范围的函数，但它仍然可以从外部调用（相对于创建闭包的 Foo() 环境而言）。一个建议是为闭合函数名添加一些特殊后缀，给直接书写调用增加一些麻烦。

闭包变量理解

闭包函数的关键是闭包变量，也就是闭包函数内所用到的外部局部变量。

其实，在一个函数内使用外部变量是很平凡的。比如：

```
let s:x = 0
function! s:Bar() " closure
    let s:x += 1
    return s:x
endfunction
```

这里只用以前的函数知识定义了一个 s:Bar() 脚本函数，它用到脚本局部变量 s:x 。每次调用 s:Bar() 时，也能递增这个变量。似乎也能达到之前闭包函数的作用，然而这只是幻觉。因为 s:x 不是专属于 s:Bar() 函数的，即使也限制了脚本作用域，也能被脚本中其他函数或语句修改。

而之前闭包函数 Bar() 的变量 x ，原是 Foo() 函数内创建的局部变量。当 Foo() 函数返回后，这个局部变量理论上要释放的，也就无从其他地方再次访问，只能通过 Bar() 这个即时定义的闭包函数才能访问。

所以，闭包变量既是外部变量，更重要的是外部的局部变量。这才能保证闭包变量对于闭包函数的专属访问。也因为这个原由，在顶层（脚本或命令）定义的函数不能指定闭包属性。如上定义 s:Bar() 函数时若加上 closure 将会直接失败。而一般只能嵌套在另一个函数中定义闭包函数，这个外层函数有的也叫工厂函数。工厂函数为闭包提供一个临时的局部环境，闭包变量先是在工厂函数中创建并初始化，而在闭包函数里面则是自动检测的，凡用到的外部局部变量都会转为闭包函数。当然了，在工厂函数或闭包函数内都可以有其他各自的普通局部变量。

在工厂函数内创建闭包函数时，闭包变量就成为了闭包函数的一个内部属性。每次调用工厂函数时，会创建闭包函数的不同副本，也就会有相应闭包变量的不同副本。也就是说，每次创建的闭包函数会维护各自的状态，互不影响。

为说明这个问明，再举个例子。比如把上面实现的递增 1 与递增 2 的两个闭包放在一个工厂函数内创建，借用列表同时返回两个闭包：

```
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()
```

另一个改动是给工厂函数传个参数，让其成为闭包递增的初值。在调用工厂函数时，也利用列表解包的语法，同时获得返回的两个闭包函数（引用）。第一次 let [Fn, X_] = FGoo(10) 用 10 作为初值，且只关心第一个闭包 Fn ，第二个 X_ 只作为占位变 量弃而不用。在执行 Fn() 数据后，第二次调用 let [X_, Gn] =

FGoo(20) 传入另 一个初值，且只取第二个闭包 Gn。然后可以发现这两个闭包能并行不悖地执行。这说明闭包变量 x 虽然是在 FGoo 中创建，却不随之保存，而是保存在各个被创建的闭包函数中。

偏包引用

自 Vim8 ，不仅为创建函数引用增加了一个全新的内置函数，而且还为 function() 与 funcref() 升级了功能。除了提供函数名外，还可以提供一个可选的列表参数，作为所引用函数的部分的参数。如此创建的函数引用叫做 partial ，这里将之称为偏包。

请看以下示例：

```
function! Full(x, y, z)
    echo 'Full called:' a:x a:y a:z
endfunction

let Part = function('Full', [3, 4])
call Part(5)
```

首先定义了一个“全”函数 Full() ，它接收三个参数，不妨把它认为是三维空间上的坐标点。假设有种需求，平面坐标已经是固定的了，只是还要经常改变高坐标。这时就可用 function() （或 funcref()）创建一个偏包，将代表固定平面坐标的前两个参数放在一个列表变量中，传给 function() 的二参数。然后调用偏包时，就不必再提供那已固定的参数，只要传入剩余参数即可。如上调用 Part(5) 就相当于调用 Full(3, 4, 5) 。

function() 的第一参数，不仅可以是函数名，也可以是其他函数引用。于是偏包的定义可以链式传递（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)
```

须要注意的是，在创建偏包时，即使只要固定一个参数，也必须写在 [] 中，作为只有一个元素的列表传入。

为什么这叫偏包，因为偏包本质上是个自动创建的闭包。例如以上为 Full() 创建的偏 包，相当于如下闭包：

```
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)
```

至于用 function() 创建通用偏包的功能，可用如下闭包模拟：

```
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

let Part = FuncPartial('Full', [3, 4])
call Part(5)
```

以上的语句 let l:arg_all = l:arg_closure + a:000 表明了在调用偏包时，传入的参数是串接在原来保存在闭包中的参数表列之后的。其实，那三条 let 语句创建的中间变量是可以取消的，只须用 return call(a:fun, a:arg + a:000) 即可。其中 a:fun 与 a:arg 变量来源于外部工厂函数 FuncPartial() 的参数，将成为闭包变量，而 a:000 则是在调用闭包函数时传入的参数。

这个 FuncPartial() 只为说明偏包与闭包之间的关系，请勿实际使用。另请注意这两概念的差别，闭包是函数，偏包是引用，偏包是对某个自动创建的闭包的引用。

创建函数引用尤其是偏包引用的 function() 与 funcref() 函数，不仅可以接收额外的列表参数，还可接收额外的字典参数。这与 call() 函数的参数意义是一样的。当需要创建引用的函数有 dict 属性时，传给 function() 的字典参数就将传给目标函数的 self ，实际上也将该字典升格为闭包变量。之后再调用所创建的偏包引用时，就不必再指定用哪个字典当作 self 了。

不过 function() 与 call() 的参数用法也有两个不同：

- call() 至少要两个参数，即使目标函数不用参数，也要传 []。function() 默 认只要一个参数即可。
- function() 可以直接传字典变量当作第二参数，不必限定第二参数必须用列表，不必用 [] 空列表作占位参数。当然也可以同时传入列表与字典参数，此时应按习惯不要改变参数位置。

lambda 表达式

lambda 表达式用于创建简短的匿名函数，其语法结构如：let Fnr = {args -> expr} 。几个要点：

- 整个 lambda 表达式放在一对大括号 {} 中，其由用箭头 -> 分成两部分。
- 箭头之前的部分是参数，类似函数参数列表，多个参数由逗号分隔，也可以无参数。无参数时箭头也不可以缺省，如 {-> expr} 形式。
- 箭头之后是一个表达式。该表达式的值就是以后调用该 lambda 时的结果。这有点像函数体，但函数体是由多个 ex 命令语句构成。lambda 的“函数体”

只能是一个表达式。

- expr 部分在使用 args 的参数时，不要加 a: 参数作用域前缀。
- 在 expr 部分中还可以使用整个 lambda 表达所处作用域内的其他变量，如此则相当于创建了一个闭包。
- 一般需要将 lambda 表达式赋值给一个函数引用变量，如此才能通过该引用调用 lambda 。也就是说 lambda 表达式自身的值类型是 v:t_func。

举个例子，假设有如下定义的函数：

```
function! Distance(point) abort
    let x = a:point[0]
    let y = a:point[1]
    return x*x + y*y
endfunction
```

这里假设用只含两个元素的列表来表示坐标上的点，该函数的功能是计算坐标点的平方和，这可作为距离原点的度量。几何上的距离定义其实是平方和再开根号，不过开根号的浮点运算效率低，尤其是相对整数坐标来说。所以在满足程序逻辑的情况下，可以先不开这个根号，比如只在最后需要显示在 UI 上才开这个根号。

然而无关背景，这个函数或许很重要，但实现很简单，实际上也可用 lambda 交待代替：

```
let Distance = {pt -> pt[0] * pt[0] + pt[1] * pt[1]}
```

当然了，这两段代码不能同时存在，因为函数引用的变量名，不能与函数名重名。分别执行这两段，测试 :echo Distance([3,4]) 能输出 25 。

前面说过，闭包函数不能在脚本（或命令行）顶层定义，但 lambda 表达式可以。因为 lambda 表达式其实是相当于创建闭包的外层工厂函数（及其调用），那当然是可以写在顶层了。不过就这个 Distance 实例，并未用到外部变量，可不必纠结是否闭包。

然后，我们利用这个函数写一个具体功能，比如计算一个三角形的最大边长。输入参数是三个点坐标，输出最大边长（的平方）：

```
function! MaxDistance(A, B, C) abort
    let [A, B, C] = [a:A, a:B, a:C]
    let e1 = [A[0] - B[0], A[1] - B[1]]
    let e2 = [A[0] - C[0], A[1] - C[1]]
    let e3 = [B[0] - C[0], B[1] - C[1]]
    let d1 = Distance(e1)
    let d2 = Distance(e2)
    let d3 = Distance(e3)
    if d1 >= d2 && d1 >= d3
        return d1
    elseif d2 >= d1 && d2 >= d3
        return d2
    else
        return d3
    endif
```

```
endfunction
```

这里，直接用单字母表示参数了，似乎有违程序变量名的取名规则。不过这也要看具体场景，因为这是解决数学问题的，直接用数学上习惯的符号取名，其实也是简洁又不失可读性的。该函数先从顶点坐标计算边向量，再对边向量调用 `Distance()` 计算距离，返回其中的最大值。

如果 `Distance` 是上面定义的函数版本，这个 `MaxDistance()` 直接可用。比如在命令行中试行：`:echo MaxDistance([2,8], [4,4], [5,10])` 将输出 `37` 。

但如果是用 lambda 表达式版本，将 `let Distance = ...` 写在全局作用域中，那么在调用 `MaxDistance()` 时再调用 `Distance()` 就会失败，指出函数未定义的错误。把这个 lambda 表达式写在 `MaxDistance()` 开头，剩余代码才能正常工作。

不过这个困惑与 lambda 无关，只是作用域规则。解析 `let d1=Distance(e1)` 时，如 果 `Distance` 不是一个函数名，就会尝试函数引用。然而在函数内的变量，缺省前缀是 `l:` ，所以它找不到在外部定义的 `g:Distance`。基于这个原因，个人非常建议在函数内部也习惯为局部变量加上 `l:` 前缀，这样就能使函数引用变量名与函数名从文本上很好地区分，避免迷惑性出错。

同时，这也说明了 lambda 的习惯用法，一般是在需要用的时候临时定义，而不是像常规函数那样预先定义。

最后提一下，lambda 作为匿名函数，vim 对其表示法是 `<lambda>123` ，与上一章介绍的字典匿名函数一样，只是在编号前再加 `<lambda>` 前缀，同时这两套编号相互独立。

小结

偏包与 lambda 表达式，本质上都是闭包，而闭包也一般只以其函数引用的形式使用。Vim8 引入这些编程概念的一个原因，是为了方便在局部环境中创建回调函数，与异步、定时器等特性良好协作。

第五章 VimL 函数进阶

5.5 自动函数

自动加载函数（`:h autoload-functions`）自 Vim7 版本就支持了。不过它涉及的机制就不仅仅是函数本身了，所以放在本章之末再讨论。其实自动加载机制已经在第一章就作为 VimL 语言的一个特点介绍过了，请回头复习一下，在那里已经将自动函数的加载流程描叙的比较细致了。

本节继续讲解有关自动函数的定义与使用。

函数未定义事件

自动加载函数的作用是，当安装的插件比较多时，不应该在启动 Vim 时全部加载（通过 vimrc `:source` 调用或放在 `plugin/` 目录下），而应只在需要用到时才加载。当然，自定义命令与映射，相当于面向用户操作的 UI，那是应该在一开始就加载好（保证有定义）。但是复杂功能的命令与映射，往往是调用函数完成实际功能的，然后所调用的函

数又可能只是个入口函数，其中又会涉及一堆相关功能的函数。那么这些函数的定义就可以延后加载，只在首次用到时触发加载，就能达到优化 Vim 启动速度的命令。

在 Vim7 版本之用，用户可以利用 FuncUndefined 这个自动事件来实现延后加载脚本的目的。例如，假设在任一 &rtp 目录下的 plug/ 中有如下脚本：

```
" >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
```

这个脚本将分两步加载。首先，由于它位于 plugin/ 子目录，故在 Vim 启动时就会读 取。在这第一次加载时，会进入 if !exists('s:load_first') 分支，该分支应该很短 ，只定义了命令与映射，并用一个 s: 变量标记已加载过一次后直接结束。关键是定义了自动事件 FuncUndefined，此后当调用了未定义函数且该函数名匹配 DT_* ，就会 重新加载这个脚本。第二次加载时，会跳过 if !exists('s:load_first') 分支，继续 加载后续代码，完成相应函数的定义。

后面那个 if exists('s:load_second') 分支，是为了避免第三次或更多次的加载。对于其他普通脚本，也可用这个机制防止重复加载，不过为仅为实现延时加载，这个分支是不必要的。一般地，如果脚本中主要是用 :function! 命令定义一些函数，重复加载也没有太大坏处，毕竟重复定义而覆盖的函数与原来的是一样。但是若脚本中需要维护某些 s: 局部变量（尤其是较复杂的字典对象）的状态，重复加载脚本就会导致这些变量的重新初始化，可能就不是想要的，这就需要避免加载。这与延时加载是两个理念，延时加载是有意地设计为加载第二次。

实际上，这延时加载的两部分，可以分别写在不同的两个脚本中。这对于非常大的脚本，可能还能进一步提高 Vim 启动速度。因为按上例，写在一个脚本中，虽然 vim 不必解释 后半部分的代码，但毕竟首先还是要打开整个脚本文件的。因些，将第一部分定义的命令、映射与自动事件放在 plugin/ 目录下，令其在 Vim 启动时就加载。第二部分的函数

定义（主体内容，长）放在另一个目录下，只要不会被 vim 在启动阶段读取就可，例如不妨就放在 autoload/ 子目录下。拆分结果示例如下：

```
" >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
```

不过在拆分时，有一行代码要注意作相应修改。就是在定义 FuncUndefined 事件时， 需要加载正确的（另一个）文件路径。上例是硬编码写入了对应的全路径。而在前面的单文件的版本中，可用 <sfile> 表示本脚本文件名。当然，在后面这个拆分版本中，若按某种规范存在相对路径中，也是可以避名硬编码的，如用以下语句代替：

```
expand('<sfile>:p:h') . '/../autoload/' . expand('<sfile>:p:t')
```

此外还须说明的是，用这种（手动）延时加载方案时，所定义的函数名最好用统一的前缀（或后缀），方便在定义 FuncUndefined 事件时指定相应的模式匹配，尽量使该匹配不扩大影响，也能保证所用函数能正确延时加载到。

自 Vim7 版本后，有了自动延时加载机制，就不必用户自己实现手动延时加载方案了。不过以上的手动延时方案，有助于理解 Vim 的自动加载机制。另外单文件版本的示例可能仍有意义，不那么复杂的脚本若不想拆分多个文件，就可按此例用 FuncUndefined 事 件实现。

自动加载函数的定义

自动加载机制，与上节讨论的拆分版的延时加载方案示例类似，不过有以下几点不同：

- 不必再写 FuncUndefined 事件；
- 将函数名前缀的 DT_ 改为 DT#
- 将定义函数的那个脚本文件名也改为 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
```

这样就可以了，不必再关注 DT#foo() 函数有没有定义，什么时刻定义，在任何地方直接使用就可以了。vim 能自动识别函数名中间包含 # 符号的函数，当作自动加载函数处理，将 # 符号之前的部分视为脚本文件名，在函数未定义时，自动到 &rtp 的 autoload/ 子目录下查找。

所以关键是要让函数名的 # 前缀与文件名保持一致，也可以不改 delaytwice.vim 的文件名，而将函数名改为 delaytwice#foo()。这种函数名的首字符允许是小写，毕竟全局函数名首字母大写的规则，主要是为了避免与内置函数冲突。

自动加载函数名中可以有多个 # 符号分隔，对应于 autoload/ 子目录下各级路径：

```
{&rtp}/autoload/sub1/sub2/filename.vim
function sub1#sub2#filename#func_name()
```

当 vim 加载含有 # 函数定义的脚本文件时，如果发现函数名前缀与文件路径不相符，就会报错，即无法顺利完成该函数的定义。不过事实上它只检查是否在 autoload/ 目 录下的相对路径，至于 autoload/ 之上的父目录是否在 &rtp 中并不强制检测。因为在正式应用环境下，该脚本文件已经是从 &rtp 中搜索到的。而另一方面，在开发测试时，你只要建立相应的目录层次，把文件扔到（某个工程） autoload/ 子目录下， 即使暂没把工程目录加到 &rtp 下，在编辑这个脚本时，也可以用 :source % 加载 当前文件进行测试，并不会因为它还不在 &rtp 中就失败。

当有了 # 函数的自动加载机制，那是否可以与 FuncUndefined 事件联用协作呢？一般情况下没有必要。但假设一种情况，如果按目前流行的方式用插件管理插件从 github 安装插件的话，一般是将每个插件放在独立的目录中，每个插件目录都加入了 &rtp中 。这样如果你真的很狂热地安装了许多插件，你的 &rtp 路径列表将变得很长。&rtp 路径在 vim 运行是至关重要，不仅这里介绍的自动加载函数，其他许多功能都要从 &rtp 中查找。如果某个插件的主要功能只是提供了 autoload/ 脚本，或许就可以尝试合并 &rtp，自己再写一个 FuncUndefined 事件，从其他地方加载脚本。

那么就要注意 # 函数内置的自动加载时机，与 FuncUndefined 事件的触发，先后关系如何，避免一些可能的冲突。下面做一个试验来探讨之。

首先，在 ~/.vim/autoload/delaytwice.vim 脚本末尾加入如下一些输入语句，用以跟踪该脚本被加载的情况：

```
function! delaytwice#foo() abort
    echo 'in delaytwice#foo()'
endfunction

" bar:
function! delaytwice#bar() abort
    echo 'in delaytwice#bar()'
endfunction
echo 'autoload/delaytwice.vim loaded'
```

然后，再自定义一个 FuncUndefined 事件：

```
execute 'autocmd FuncUndefined *#*  call MyAutoFunc()'
```

```
function! MyAutoFunc() abort
    echo 'in MyAutoFunc()'
    " TODO:
endfunction
```

它也匹配任何中间含 # 符号的函数名，但假设它们（有些）没放在 &rtp 中，所以需要写个入口函数从其他地方查找并加载定义文件。将该代码放在 plugin/ 下某个文 件中，便于在每次启动 vim 时自动执行，保证该事件已定义。

接下来就可以测试了，重启 vim （或打开 vim 的另一个实例会话），在命令行执行如下命令，其输入也附于其后：

```
: call delaytwice#foo()
autoload/delaytwice.vim loaded
in delaytwice#foo()
```

这说明只触发了 vim 内置的自动加载机制，它自动加载了 delaytwice.vim 文件，然 后 delaytwice#foo() 函数就是已定义了，就可调用该函数了，不会再触发 FuncUndefined 事件。

再重启一个 vim ，在命令行调用一个在该文件中并不存在的函数，比如将 foo() 小写 误写成了大写 Foo()，其输出如下：

```
: call delaytwice#Foo()
autoload/delaytwice.vim loaded
in MyAutoFunc()
autoload/delaytwice.vim loaded
E117: Unknown function: delaytwice#Foo
```

从结果可分析出，vim 仍是先按自动加载机制，找到 delaytwice.vim 并加载，然后再尝试调用 delaytwice#Foo()，它仍是个未定义函数。这二次调用时，才触发 FuncUndefined 事件。当然我们这里自定义的 MyAutoFunc() 并没做实际工作，并不能解决函数未定义问题。于是 vim 再按自动加载机制，找到并加载 delaytwice.vim。加载两次后仍未解决问题，vim 就报错了。

当然了，如果在 &rtp 中并没有找到 delaytwice.vim 或者调用 :call nofile#foo() ，它只出输出 in MyAutoFunc() 这行以及错误行。但它显然是遍历过一次 &rtp 未找到相应文件，才触发 FuncUndefined 事件的。

现在，又假设不自定义 FuncUndefined 事件与 MyAutoFunc() 处理函数，只按 vim 的自动加载机制，如果调用了在自动加载文件中其实并未定义的函数，会是什么情况呢：

```
: 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
```

可见，第一次调用 delaytwice#Foo() 时，加载了两次脚本，其内的 delaytwice#foo() 与 delaytwice#bar() 就是已定义的，可正常使用了。然后每次 误用 delaytwice#Foo() 或 delaytwice#Bar() 都会再触发加载一次脚本。

综上，可得到如下结论：

- vim 会记录已加载的脚本文件，当调用自动加载函数时，若分析自动加载函数所对应的自动加载脚本并未加载，就会先搜索并加载相应的脚本，再次调用原函数。
- 在自动加载脚本已加载或未找到相应脚本的情况下，调用未定义的自动加载函数才会触发 FuncUndefined 事件，会先调用自定义的事件处理函数，若无法触发，再次搜索加载相应的脚本。

自动函数与其他函数的比较

首先，要明确一件事，自动加载函数是在全局作用域的。也就是相当于全局函数，可以在任何地方使用。

但是，它又有某些局部函数的作用。比如，可以在两个自动加载脚本中定义“相同”的函数，onefile#foo() 与 another#foo()。但实际上它们仍是两个不同的函数，因为包含 # 在内的整个字符串 onefile#foo 与 another#foo 才是它们的函数名。它们只是名字上包含相同后缀的相似函数而已。仅管如此，能在不同文件（插件）中，利用相同的词根定义相同（或相似）功能的不同实现，也是很有意义的，增加代码可读性与维护。

在为自动函数定义函数引用时，也要使用其全名。同时，自动加载函数它是函数，不是变量。所以，在定义 onefile#foo() 的 onefile.vim 文件中，还可以定义 s:foo 变量。但最好不要这样增加混乱，除非将其定义为同名函数的引用，如 :let s:foo = function('onefile#foo')。因为即使在同一个文件中，也必须使用包含 # 的函数全 名，它可能很长，使用不太方便，所以定义一个局部于 s: 的函数引用，是有意义的。

除了函数名可以加 # 符号实现自动加载机制，全局变量名也可以加 # 符号。但是只有当这个变量用于右值，如 :let default = g:onefile#default 才会触发搜索加载相应脚本（onefile.vim）。但用于左值，如 :let g:onefile#default = 5 却并不会 触发加载脚本。这个区别其实是为了让用户在触发加载 onefile.vim 之前，就能设置 g:onefile#default 的值，那可作为相关插件的用户配置变量，比之前惯用的 g:onefile_default 变量名似乎更有意义，更像 VimL 风格。

正因为 # 符号也可用于变量名，才千万注意不要将一个函数引用变量保存在 # 变量 名中（包括 lambda 表达式），虽然那是合法的，但非常不建议如此混乱。只有真的有意设计开放给用户的重要配置才定义为 # 全局变量，并始终加上 g: 前缀。而函数名是不能加 g: 前缀的，如此容易直观地区分。

总之，自动加载函数是 VimL 中一个极优秀的设计。如果说函数引用是注重从内部管理，那么自动加载函数则注重从外部管理。善加利用，可极大增强 VimL 代码的健壮性与可维护性。若说有什么缺点的话，那就是自动加载函数名可能太太太长了。并且要由用户来保证函数名前缀与文件名路径的一致性，如果脚本文件改名了，或移动了路径层次，手动修

改函数名也是一大问题。非常期望在后续版本中能增加什么语法糖，能使得在本文件中定义与使用自动加载函数更加简洁些。

第六章 VimL 内建函数使用

一般实用的语言包括语法与标准库，毕竟写程序不能完全从零开始，须站在他人的基石之上。而要开发更有产品价值的程序，更要站在巨人的肩膀上，比如社区提供的第三方库。

细思起来，VimL 语言的“标准库”包括两大类：内建命令与内建函数。用户在此基础上可自定义命令与自定义函数，再合乎语法地组成起来，以达成所需的功能。第三章简要地介绍了部分基础命令，其实那更倾向于 Vim 编辑器的功能。本章要介绍的内建函数，则更倾向于 VimL 语言的功能。

不过本章将会是比较无聊的一章。帮助文档 `:help function-list` 会按类别列出内置函数，`:help functions` 则会按字母序列出内置函数，可供参考。中文用户可找一份帮助文档的中译本，虽然可能不是最新版本的，不过绝大部分内置函数都应该是稳定向下兼容的。

所以本章不会（也没必要）罗列所有内建函数，只择要讲些内建函数的使用经验技法。要查看某个函数的解释，请直接 `:help func_name()`，请注意加一对括号，限定查函数的文档，否则有可能查到的是同名的命令或选项等。

6.1 操作数据类型

字符串运算

Vim 是文本编辑器，所以处理文本字符串是一重点任务。每个字符在计算机内部都用一个整数表示（即编码值，与用数字字符组成表示的可读整数不同概念），具体如何对应取决于编码系统（有时简称编码）。目前计算机界的趋势是用 `utf-8` 编码表示 Unicode 字符集，因为它与最早的 ASCII 编码兼容。一个汉字在此编码下用 3 个字节表示，英文字符仍用一个字节表示。

- nr2char() 将编码值转为字符
- char2nr() 将字符转为编码值
- str2nr() 将字符串转为整数
- str2float() 将字符串转为浮点数

由于 VimL 没有字符串类型，`char2nr()` 其实是将字符串首字符将为编码值的。默认按 `utf-8` 编码获取编码值与字符的对应，但可传入额外参数按其他编码系统对应。例如：

```
: echo char2nr(' ') |" --> 20013
: echo nr2char(20013)  |" -->
```

整数类型与字符串一般可自动转换，一般用不上 `str2nr()`。但如果从安全考虑习惯主动判断类型的话，要注意从命令行输入的参数都是字符串，不是整数。此外，字符串不会自动转为浮点数，而是截断为整数，所以确实要处理浮点数是，用 `str2float()` 转换。

- printf() 格式化字符串

简单的字符串连接用连接操作符 `.` 即可。要组装复杂字符串时可用 `printf()` 函数，通过字符串模式与 `%` 占位符，插入变量。其用法与 C 语言的 `sprintf()` 类似，因为该函数会返回结果字符串，“打印”字符串用 `:echo` 命令。

- escape() 将字符串中指定的字符用反斜杠 \ 转义
- shellescape() 转义特殊字符以适于 shell 命令
- fnameescape() 转义特殊字符以短于 Vim 命令，主要用于转义文件名参数

当组装字符串用于当作命令执行时，为安全起见，应先调用 `shellescape()` 或 `fnameescape()` 进行转义。这两个函数自有其转义策略，适用大部分情况。当有特殊需求时，可用 `escape()` 指定要转义哪些字符（传入第二参数的字符串中出现的所有字符都表示要转义的）。

- tolower() 将字符中转为小写
- toupper() 将字符串转为大写
- tr() 按一一对应的方式转换字符串
- strtrans() 将字符串转换为可打印字符串

`tr()` 函数进行简单的字符串转换（不是正则替换），效果如同 unix 工具 `tr`。大小写转换是其一种特例策略，如转大写相当于 `tr(str, 'abcdefg...', 'ABCDEFG...')`。而 `strtrans()` 是按 vim 的自定策略将不可打印字符转换为可视字符（组合，一般以 ^ 开关）表示。

- strlen() 按字节数获取字符串长度
- strchars() 按字符数获取字符串长度，当含宽字符（如汉字时）与字节长度有差异
- strwidth() 字符串宽度，显示在屏幕上时将占用的列宽度，但未处理制表符
- strdisplaywidth() 字符串实际显示宽度，并按设置处理制表符宽度
- byteidx() 第几个字符的字节索引，不单独处理组合字符
- byteidxcomp() 也是字符索引转为字节索引，组合字符单独处理

字符串可像列表一样用中括号索引，那是按字节索引的。当字符串中存在宽字符时，字符数与字节数不一致，这就需要处理字符索引与字节索引的不同。请仔细观察以下示例：

```
: let str = 'vim  '
: echo strlen(str)    |" --> 10
: echo strchars(str)  |" --> 6 vim
: echo len(str)       |" --> 10
: echo strwidth(str)  |" --> 8
: echo str[0:2]       |" --> vim
: echo str[4]         |" --> <e4>
: echo str[4:5]       |" --> <e4><b8>
: echo str[4:6]       |" -->
```

组合字符（有的书籍叫重音字符），中国人一般不必关注，欧洲人才用得到。比如 é 是通过一个正常的 e 字母加上重音符组成而成的（`'e' . nr2char(0x301)`），显示上像是一个字符，但计算机要用两个字符表示。至于算一个字符还是两个字符，似乎都有理有用，所以就提供了不同的函数或可选参数来处理这种情况。这与汉字宽字符的情况不一样。汉字的三个字节是不可分的，取第一字节是无效字符。但组合字符的第一字节仍是个有效字符（字母）。

字符与编码看似简单，如同空气与水一样简单，但深入细处还挺复杂。所以建议初学者不必深究，始终用英文文本示例测试学习即可。当实际工作中遇到中文问题时再回头查阅。此外，据说早期的中国程序员常要念经“一个汉字等于两个字节”，那是用 GB 编码的原因 ，现在请升级经文“一个汉字等于三个字节”。

- stridx() 查找一个短字符串在另一个长字符串第一次出现的起始索引
- strridx() 查找一个短字符串在另一个长字符串最后一次出现的起始索引
- strpart() 截取字符串从某个索引开始的定长子串

查找简单子串存在情况可用 `stridx()`，要求精确匹配，且大小写敏感。返回的结果索引是字节索引，索引从 0 开始，若不存在子串返回 -1。截取子串可用中括号索引切片方式，如上例 `str[4:6]`，参数是起始索引与终止索引（含双端）。而 `strpart()` 的参 数是起始索引与长度，如上例等效于 `strpart(str, 4, 3)`。

- match() 查找一个正则表达式在字符串出现的起始索引，不匹配时返回 -1
- matchend() 查找一个正则表达式在字符串出现的终止索引
- matchstr() 返回字符串中匹配正则表达式的部分，不匹配时返回空串
- matchlist() 将正则匹配结果按分组返回至列表中，不匹配时返回空列表
- substitute() 正则表达式替换，:s 命令的函数式
- submatch() 获取正则匹配的分组子串，只可用于 :s 命令的替换部分

这几个函数用于处理正式表达式的匹配查找与替换。如果仅是要判断是否匹配，可直接用操作符 `if str =~# pattern`。`match()` 函数主要是还能返回匹配成功的起始索引，相应地 `matchend()` 返回的是终止索引。`matchstr()` 返回的是匹配到的整个子串。如果正则表达式中有括号分组 `\(\)`，最好用 `matchlist()` 函数，它返回的列表中，第一个元素（`[0]`）就是匹配到的整个子串，其后是按顺序的分组子串。其中的关系可用如下伪代码表示：

```
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)
    slist[0] == matchstr(s, p) == & == submatch(0)
    slist[1] == \1 == submatch(1)
    slist[2] == \2 == submatch(2)
    ...
endif
```

替换函数 `substitute()` 的参数及意义与 `:substitute` 命令的几个部分完全一样。不过命令可以缩写为 `:s`，函数不可以缩写。如以下两个语句功能类似：

```
: s/pat/sub/flag |"     pat   sub
: call substitute(line('.'), pat, sub, flag)
```

在替换部分 `{sub}` 可用表达式，以 `\=` 开始即可，如此 `submatch()` 表示前面 `{pat}` 部分的分组子串。而在以常规字面字符串表示 `{sub}` 部分时，则用 `\1` 表

示分组子串。

- string() 将其他任意变量或表达式转为字符串表达，类似 :echo 的显示
- expand() 将具有特殊意义的标记（如 % # <cword> 等）展开
- iconv() 转换字符串编码
- repeat() 将字符串重复串接多次生成长字符串
- eval() 将字符串当作表达式来执行，并返回结果
- execute() 将字符串当作命令来执行，将结果返回为字符串

注意，eval() 与 execute() 很灵活，但比较低效，也可能有一定风险。如有其他更优雅的实现写法，尽量用替代方案。

浮点数学运算

用 VimL 做数学运算并不常见，但如果啥时想到需要她，她也在那儿。一般整数运算直接用操作符，浮点运算才需要调用函数，且这些内置函数的结果一般也是浮点数，即使参数都是整数。

- float2nr() 将浮点数转为整数类型
- trunc() 截断取整
- round() 四舍五入取整
- floor() 向下取整
- ceil() 向上取整

这几个函数都是取整运算，但实际上只有 float2nr() 的结果是整数类型（v:t_number），其他函数取整后仍为浮点数（v:t_float）。float2nr() 与 trunc() 意义一样是截断取整。当涉及负数，取整可能不太直观，请看示例：

```
: 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
```

当四舍五入正好在中值时（如小数部分是 0.5），取远离 0 那个整数，类似：

```
: round(+float) == trunc(+float + 0.5) |" -->
: round(-float) == trunc(-float - 0.5) |" -->
```

- fmod() 取余数
- pow() 取幂
- sqrt() 开平方

整数取余数可直接用操作符 %，该操作符不能用于浮点数。fmod() 可用于浮点数的取余，即使整数也当作浮点处理。VimL 并没有整数取幂的操作符（其他语言有用 ^ 或 ** 作幂运算的），须用 pow() 函数求幂，结果也总是浮点数；

```
echo 10 % 3          |" --> 1
echo fmod(10, 3)     |" --> 1.0
echo pow(2, 10)      |" --> 1024.0
echo pow(4, 1/2)     |" --> 1.0    1/2 = 0
```

```
echo pow(4, 1/2.0)   |" --> 2.0
echo sqrt(4)         |" --> 2.0
```

- exp() 自然指数
- log() 自然对数，以 e = 2.718282 为底
- log10() 常用对数，以 10 为底
- 三角函数与反三角函数：sin() cos() asin() acos() 等

log() 是 exp() 的反函数，在数学上的记号是 ln；而数学上记为 lg 的对数，程序上是 log10()。这个命令习惯应该是源自 C 语言的标准库函数。同样，一众三角函数也是类似 C 语言的，参数是以弧度单位表示的角度。不过很难想像需要在 VimL 中 用到这些略为高深的数学计算的场景。

- isnan() 判断是否为非数

自 Vim8 引入非数的判断。像 0/0 这样的计算结果叫非数，在其他一些计算机语言与文档中习惯用 NaN 来表示。可能为了更好地与其他数据文件交互，Vim 也增加这个函数来处理非数。

列表与字典运算

在第四章介绍数据结构时，已经顺便介绍了操作列表与字典的函数。这里不再重复，只作些补充说明。

首先，很多函数可同时作用于列表与字典，甚至字符串。因为脚本语言弱类型的缘故，没法限定传入函数的参数，只能根据参数类型作出不同的合理反馈。例如：

- len() 取列表或字典集合中元素个数，也取字符串的（字节）长度
- empty() 可判断是否空列表、空字典或空字符串，整数 0 也认为是空的
- match() 还能匹配字符串列表，返回能匹配成功的元素索引

其次，列表与字典都是集合，有一类高阶函数，可接收另一个函数（引用）作为参数，用于处理集合内的每一个元素。比如 map() 与 filter() 函数。

前面提及，VimL 有许多与命令同名的函数，都是实现类似的功能。但 map() 是例外， 它与定义键映射的 :map 命令没有语义关系，完全是不同的概念。

- map({expr1}, {expr2}) 修改集合的每个元素

其中参数一 {expr1} 可以是列表如字典，参数二 {expr2} 是函数引用。参数一集合的每个元素，传给参数二所代表的函数，将结果值替换原来的元素。最终会原位修改列表或字典。关键是参数二所引用的函数定义要遵循一定的规范，它应接收两个参数，map() 会将每个元素的索引与值传给该函数（字典元素的索引即是键名）。整个流程可如下模拟：

```
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
```

如果处理每个元素的函数很简单，则可不必创建函数再传入函数引用。可用一个字符串代替，该字符串调用 eval() 执行后，将结果替换原元素。在字符串用，用内置变量 v:val 代表迭代的每个元素值，v:key 代表元素索引（键名）。相当于传入函数版本的两个参数。用这种方法得注意字符串的转义，建议用单引号括起字面字符串。可用其他字符串函数或操作符组装，最终结果的字符串再调用 eval() 计算结果新值。

事实上，在低版本的 vim 中， map() 函数的参数二只能用字符串。这才需要 v:key 与 v:val 这两个特殊变量标记置于可执行字符串中。自 vim8 后，强烈建议使用函数引用参数。这就无须理解 v:key 与 v:val 的即时意义。不过在定义处理元素的函数时，建议也用 key 与 val 作为函数形参，这使整个代码的可读性更佳：

```
function! MapHandle(key, val) abort
    let l:result = deal with a:key and a:val
    return l:result
endfunction
```

如果处理函数很简单，也可不必预定义函数，即时定义 lambda 也可以，因为 lambda 表 达式的值也正是一个函数引用。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
```

以上示例也说明了 map() 函数是原位修改的，如果不想修改原集合，可先调用 copy() 创建副本。低版本中等效的用字符串调用方式如下：

```
echo map([1, 2, 3], 'v:val * 2')
```

像这样简单的功能，也许字符串方式写来更简洁，但稍为复杂的功能，可执行字符串的表示法就可能比较费解了。比如要将原列表中每个元素加上尖括号 <> 括起来，以下三种调用方式都能实现：

```
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 表达式可避免多重引号的理解困难。而且 lambda 表达式或函数若预先定义的话，在其他地方也是可用的。而含 v:val 的特征字符串，放在其他地方几乎是没什么

意义了。

- filter({expr1}, {expr2}) 过滤集合内的元素

filter() 与 map() 函数类似。参数二所代表的处理函数，也接收索引与值两个参数，但是要求返回布而逻辑值。如果返回的是真（数字 1），则保留不处理，如果返回的是假（数字 0），则删除相应的元素。模拟流程如下：

```
function! FilterDict(dict, fun)
    for [l:key, l:val] in items(a:dict)
        let l:bKeep = a:fun(l:key, l:val)
        if empty(l:bKeep)
            unlet a:dict[l:key]
        endif
    endfor
endfunction
```

自己模拟过滤列表可能略有麻烦，因为如果正向迭代，删除元素后，索引可能会变化。当然了，你不必真的自己写或用这样的模拟函数，请用内置的库函数！

同样地，可以用字符串或 lambda 表达式。且在支持 lambda 表达式的 vim 中，尽量用 lambda 表达式。

- sort({list} [, {fun}, {self}]) 为列表排序，从小到大
- uniq({list} [, {fun}, {self}]) 删除相邻重复元素

几乎在任一本算法教科书，排序都是重点。但是几乎在任一个语言中，排序都有已优化实现的库函数，不必自己写的，自己需要做的只是提供比较函数，说明要如何排序的需求。VimL 要求的比较函数能接收两个参数，返回值意义如下：

- 0 两个参数视为相等
- 1 第一个参数视为比第二个参数大
- -1 第一个参数视为比第二个参数小

sort() 函数只能为列表排序，因为字典是无序的。第二参数 {fun} 一般是函数引用，可用 lambda 表达式，但不支持像 map() 那样的可执行字符串。然而，可以是普通字符用于表示 vim 预设的几种排序策略（常用需求）：

- 空串或省略，按字符串排序，类似 :sort 命令为当前 buffer 的排序行为。
- l 或 i，忽略大小写的排序
- n 按数字排序，非数字类型的元素认为是 0
- N 按数字排序，字符串会转为数字
- f 按数字排序，列表元素限定仅是数字或浮点数

VimL 的 sort() 是稳定排序算法，即如果两个元素相等（按 {fun} 返回 0），排序后它们也保持原来的相对顺序。如果 {fun} 参数是含 dict 属性的函数，则要提供第三参数 {self}，一个作为 self 的字典变量。

uniq() 函数的参数用法与 sort() 相同。且一般应该对已排序的列表调用 uniq()，因为它只比较相邻元素而去重。

小结

VimL 的标量主要就是字符串与数字，集合也就列表与字典。所以为这些数据类型提供了大量的库函数 api。用 :h type() 查看支持的所有变量类型。但其他类型需要支持的操作非常有限，故无必要有什么专门函数处理。

第六章 VimL 内建函数使用

6.2 操作编辑对象

与 Vim 可视编辑的有关的几个概念对象是缓冲（buffer）、窗口（window）与标签页（tabpage），还有目前较少用到的在命令行参数提供的文件列表（argument list）。VimL 也提供了许多函数以供脚本来控制这些编辑对象。

编辑对象背景知识

很早期的 vi 一次只能编辑一个文件。不过从命令行启动时可以提供多个文件名参数，首先编辑第一个文件，编辑完后可以接着编辑下一个文件。如以下命令启动：

```
$ vim file1 file2 file3
```

Vim 就记忆着这三个文件，称之为参数列表，相当于执行了如下 VimL 语句：

```
: let arglist = ['file1', 'file2', 'file3']
```

注意在 Vim 启动时，还可以加很多命令行选项（以 - 开头的参数），一般用于指定 Vim 以何种方式、何种配置等启动。这些选项在 Vim 启动过程中就会被处理掉，不会保存在参数列表中，所以参数列表只保存待编辑的文件名。

后来，Vim 支持同时编辑多个文件。作为通用编辑器，配置好 vimrc 后，它也经常省略命令行参数，直接以裸命令 $ vim 启动，其参数列表就为空 []。然后在 Vim 自 己的命令行中用命令 :edit file 打开要编辑的文件。

Vim 每打开一个文件，就创建一个缓冲（buffer），并记录相应的缓冲信息。即使打开另一个文件，曾经打开的而目前看不见的文件，也记忆着它的缓冲，除非用命令显示地清除它。Vim 的这个缓冲概念与系统缓存并不一样，对于非活跃 buffer （看不见的文件），Vim 也不可能将文件的所以内容留在内存中，尤其是打开了很多大文件。可认为 buffer 是 Vim 为每个编辑文件创建的一个对象，记录着一些必要的信息。但是，也不一定每个 buffer 都对应着文件系统内的物理文件（磁盘上的文件），例如新建 buffer 尚未保存 甚至未命名，还有很多标准插件与三方插件的辅助窗口中的特殊 buffer 根本就不想写入 文件。

然后正在编辑的活跃 buffer 必然是显示在窗口的。早期的 vi/vim 也只支持一个窗口，后来实现了多窗口。一个窗口只能装载一个 buffer，但一个 buffer 可以同时显示在多个窗口中。再后来更扩展到多个标签页，每个标签页都可以分隔为多个窗口。

缓冲、窗口与标签页都被 Vim 顺序编号以便维护，这有点像参数文件列表的索引。不过参数列表是真当作列表变量类型的，索引从 0 开始。而缓冲、窗口与标签页的编号都从 1 开始。关闭文件并不意味着关闭缓冲，即使清除缓冲或隐藏缓冲也不会改变每个缓冲的编号，但是关闭或移动窗口（或标签页），却会改变它们的编号。为此，自

Vim8 起，又引入窗口 id 概念，它是唯一且稳定的（不过似乎尚未有标签页 id 的概念）。

然而要指出，即使引入了缓冲概念，参数列表也还是有价值的。在有些情况下启动 vim 确实有明确目标要编辑某系列文件，将所有文件保存在参数列表中（其实在进入 vim 后 也可以提供或更改文件参数列表），就有很多批量命令能统一处理这系列文件。即使引入窗口 id 概念，也还有窗口编号的价值。因为窗口编号更直观，从左到右，从上到下，很容易知道哪个窗口是 1 2 3 4 。

查看缓冲可用如下命令之一：

```
: buffers
: ls
```

注意 :buffers 是有 s 后缀的复数形式，那才是打印缓冲列表的意思。如果是单数命令 :buffer 则一般需要接个参数，用于打开另一个缓冲的意思。:ls 更简短，这命令在 shell 中是列出文件意思，而在 Vim 中是列出缓冲的意思。这两个命令的输出中，包含缓冲编号及相应的文件名等信息。

在任一时刻，都有（正在编辑中的）当前缓冲，当前窗口与当前标签页的概念。如果提供了参数文件列表，也有当前文件的概念。不过当前文件不一定与当前缓冲相同。因为常规编辑命令不会改变参数列表，你可以用 :e 或 :b 命令切换到编辑另一个可能并不在参数列表中的“无关”文件，但在 vim 内部随参数文件列表保存的当前索引并不会改变。

最后，将这三个或四个概念统称为编辑对象。当了解这些编辑对象的意义后，就能更好地理解相关的函数功能了。初学者可能会对缓冲与文件（参数列表）有所迷惑，日常使用时可不求甚解认为缓冲即是指文件。不过编程时需要准确理解其中的不同。

获取编辑对象信息

- bufnr() 获取缓冲编号
- bufname() 获取缓冲名字
- winnr() 当前窗口编号，winnr('$') 获取窗口数量即最大编号
- tabpagenr() 当前标页面编号，'$' 参数获取标签页数量
- tabpagewinnr() 某个标签页的当前窗口编号
- bufwinnr() 获取某个缓冲的窗口编号
- winbufnr() 获取某个窗口的缓冲编号

其中，bufnr() 与 bufname() 的参数是一样意义，指示如何搜索一个缓冲，搜索失败时前者返回 -1，后者返回空字符串：

- 数字：即表示缓冲编号。bufnr(nr) 一般返回编号本身（无效时返回 -1）。 bufname(nr) 用于获取指定编号的缓冲文件名。
- 字符串：除了以下特殊字符意义，将该字符串当作文件名模式去搜索缓冲，也就是说不必指定文件全名，可以按文件名通配符（不同于正则表达式）搜索。但如果有歧义能匹配多个，或未能匹配，都算失败，返回 -1 或空串。当然会优先匹配全名，如果要限定只当作全名匹配，可加前后缀 ^ 与 $。
- 缺省：不能缺省参数，至少提供空字符串。
- ""：空字符串表示当前缓冲。
- "%"：也表示当前缓冲。

- "#"：表示另一个轮换缓冲（在编辑当前缓冲之前的那个缓冲）。
- 0：数字零也表示另一个缓冲。

注意，bufname() 返回的缓冲名，与 :ls 命令输出的相应缓冲行的主体部分相同。该缓冲名是否包含文件全路径名，可能与当前路径有关。所以，如果要在程序中唯一确定一个缓冲，应该用 bufnr() 的返回值，bufname() 一般只用于显示。

bufnr() 函数不能无参数调用，空字符串或数字零都是有特殊意义的参数。但是，winnr() 与 tabpagenr() 一般是无参数调用，以获取当前窗口（标签页）的编号，而用 "$" 参数表示获取最后一后窗口（标签页）编号，也就是最大编号或其总数量。tabpagewinnr() 用于获取另一个标签页的当前窗口编号，比 winnr() 多加一个标签页编号参数在前面。因为每个标签页都有当前窗口的概念，即是最后驻留的那个窗口。此外，"#" 参数可用于 winnr() 与 tabpagewinnr() 表示之前窗口编号（即进入当前窗口之前的那个窗口，<C-w>p 或 :wincmd p 将进入的窗口）；但不可用于 tabpagenr() 函数，因为 Vim 似乎没有维护之前标签页的概念。

以窗口编号为例，其典型调用方式小结如下： * winnr() 当前窗口编号 * winnr('$') 最大窗口编号或窗口数量 * winnr('#') 之前窗口的编号

因为一个缓冲可能显示在多个窗口中，所以 bufwinnr() 返回的是显示了指定缓冲的第一个窗口编号。其参数与 bufnr()意义相同，可认为先调用 bufnr() 确定缓冲编号再查找相应窗口编号。反之，一个窗口在一个时刻只显示一个缓冲，所以 winbufnr() 返回的缓冲编号是确定的。其参数是窗口编号，用 0 表示当前窗口，但不能像 winnr() 那样使用 $ 或 # 字符表示特殊窗口，否则字符串按 VimL 自动转换规则转为数字 0，仍是调用 winbufnr(0)。

- bufexists() 检测一个缓冲是否存在
- buflisted() 检测一个缓冲是否能列表出来（:ls）
- bufloaded() 检测一个缓冲是否已加载
- tabpagebuflist() 返回显示在某个标签页中的所有缓冲编号列表

以上三个检测缓冲状态的函数，所以接收的参数除了缓冲编号外，若字符必须是文件全名（全路径或相对当前路径），并能像 bufnr() 的参数那样支持文件通配符。一些特殊缓冲并不会被列表出来，取决于局部选项 &buflisted 的设置。已加载的缓冲是指显示在某个窗口的缓冲，但如果一个缓冲设置了 &bufhidden 局部选项为可隐藏 hide， 则它即使不显示了也仍算加载状态。

若要获取所有已显示在窗口中的缓冲，可用 tabpagebuflist() 函数，它返回一个列表，收集了指定标签页中所有窗口内显示的缓冲（编号）；缺省参数时指当前标签页。在所有标签页中显示的缓冲都是已加载状态（但已载缓冲可能还包含一些隐藏缓冲），如下函数可返回几乎所有已加载缓冲的列表：

```
function! BufLoaded() abort
    let l:lsBufShow = []
    for i in range(1, tabpagenr('$'))
          call extend(l:lsBufShow, tabpagebuflist(i))
    endfor
    return l:lsBufShow
endfunction
```

- argc() 参数文件列表个数
- argv() 参数文件列表，或返回指定索引的文件参数
- argidx() 当前所处参数文件的索引
- arglistid() 返回参数文件列表的ID

这是几个处理参数文件列表的函数。在去除启动选项后，argc() 与 argv() 就是命 令行参数。无参数调用 argv() 返回整个文件列表，但可指定索引 argv(idx) 返回 相应的文件名，argc() 就是这个列表的长度，即文件个数，而 argidx() 是指所谓 的当前文件的索引。但是 Vim 还对参数文件列表作了扩展，除了从命令行启动时指定的参数列表叫做全局参数文件列表外，还可以为每个窗口定义局部参数文件列表，所以有了 arglistid(winnr, tabnr) 函数用以返回某个指定窗口（参数都可缺省，即用当前窗口或当前标签页）的参数文件列表，全局的参数文件列表 ID 用 0 表示。

- win_getid() 获取指定标签页与窗口编号（可缺省默认当前）的窗口ID
- win_gotoid() 切换到指定窗口ID的窗口，有可能切换当前标签页
- win_id2win() 将窗口ID转换为窗口编号，只在本标签页查找
- win_id2tabwin() 将窗口ID转换为二元组 [标签页编号, 窗口编号]
- win_findbuf() 根据缓冲编号查找所有相应的窗口ID（是列表类型）

这几个处理 window-ID 的函数是从Vim8 版本引入的。函数名已经很望文生义了，可以在窗口ID与窗口编号（及标签页编号）之间互相转换。要注意的是，每个标签页的窗口编号都是从 1 开始重新编号，相互独立。但窗口ID是全局的，所有标签页的窗口共享一套统一的ID。

获取编辑对象数据

前文在介绍 VimL 变量作用域时，提到三个特殊的局部作用域前缀 b: w: t: ，那就是分别保存在特定缓冲、窗口与标签页的变量。如果仅用这个前缀，而无后缀主体变量名，那就是表示收集了所有相应局部变量的字典（如 b: 也是个类似 s: 的特殊字典 ）。从语义上理解，字典可当作一个对象，键当作属性。那么这些局部变量也就相当于相应编辑对象的属性数据了。以下的 get/set 函数就是处理这些变量的函数：

- getbufvar() 返回缓冲局部变量 b:
- setbufvar() 设置缓冲局部变量的值
- getwinvar() 返回窗口局部变量 w: （限当前标签页）
- setwinvar() 设置窗口局部变量的值
- gettabvar() 返回标签页局部变量 t:
- settabvar() 设置标签页局部变量的值
- gettabwinvar() 返回窗口局部变量 w:
- settabwinvar() 设置窗口局部变量的值

以缓冲局部变量为例，函数参数原型是 getbufvar(, ,)。其参数一是缓冲编号或名字（类似 bufnr()的参数意义）；参数二的变量名是没有 b: 前缀的 主体名字，即 b: 字典的键；参数三是默认值，当不存在相应变量时的返回值，该参数可缺省，缺省时就是空字符串，即当变量不存在时也不会出错，而至少返回空字符串。参数二变量名不可缺省，当它是空（字符串）时，返回 b: 字典本身。设值函数参数原型时 setbufvar( )，第三参数不可缺省。

窗口局部变量取值与设值函数，可能与标签页有关。gettabwinvar(

)，需要在第一个参数前多插入一个标签页编号，如果取当前标签页的窗口变量，则用 getwinvar( )。窗口编号参数传 0 的话，表示当前窗口。

- getbufinfo() 返回缓冲对象信息列表
- getwininfo() 返回窗口对象信息列表
- gettabinfo() 返回窗口对象信息列表

这三个函数是从 Vim8 引入的。其返回类型是字典的列表，即每个列表元素都是字典，字典所包含的属性键依对象而不同。如果参数限定了一个对象，返回值也是包含一个元素的列表；如果根据参数无法确定（搜索到）任一对象，则返回空列表。如果没有参数，则返回由所有对象的信息字典组成的列表。

如果提供参数，getwininfo() 需传入窗口ID，而 gettabinfo() 传入标签页编号。而 getbufinfo() 稍为复杂，除了可像 bufnr() 那样传入缓冲编号或名字外，还可以用字典指定筛选缓冲的条件：buflisted 已列出的，bufloaded 已加载的。

这三个函数返回的对象信息字典，详细的键名解释请参考文档。但是都有一个键 variables （注意单词复数形式），其值是另一个字典（引用），即是特殊字典 b: 或 w: t:。所以 get...info() 函数也实现了 get...var() 的功能，不过前者所得信息大而全，用法更复杂。另外 get...var() 函数可获取局部选项的值，以 & 为前缀的变量名传入即可，但这无法由 get...info() 获得，因为选项值并不保存在 b: 字典中。

获取光标位置信息

显然，当前光标只有一个确定位置。但 Vim 另有一个光标标记（mark）的概念，用于记忆多个位置信息。例如在普通模式下用 mx 命令，就定义了标记 x，保存着当前光标的位置。此后移动到他处后，再用命令 'x（单引用）就能跳回标记 x 的行首， 使用 \x x a-z Vim : :‘<,’>‘，那就分别表示选区起始行与终止行的标记。

- line() 光标或标记的行号
- col() 光标或标记的列号（字节索引）
- virtcol() 光标或标记的屏幕占位列号
- winline() 光标在当前窗口的行号
- wincol() 光标在当前窗口的列号
- screenrow() 光标在屏幕的行号
- screencol() 光标在屏幕的列号

以上 line() col() 返回的行列号是相当缓冲文件而言。col() 是按字节列号的，第一列是 1，0 用于表示错误列号。virtcol() 指屏幕占位列号，光标所在字符所占的最后一列。假如一行全是汉字，光标停在第四个汉字上，col() 是 10，因为前三个汉字只 9 字节，第四汉字从第 10 字节开始；virtcol() 是 8，因为每个汉字占两列宽，第四个汉字已占到第 8 列。当有制表符 \t 时，屏幕列与字节显然也是不同的。不过这三个函数必须带参数调用，字符串参数意义如下：

- . 单点号表示当前光标
- $ 当前行最后一列
- 'x 表示 x 标记

- v 用于选择模式下，表示选区起始（因当前光标只表示选区终止）

特殊用法是 col([, '$']) 可获得指定行的最后一列。

winline() 与 wincol() 不带参数，只用于获取当前光标相对于窗口的行列号，因为标记位置可能不在窗口显示区域，为标记调用这两个函数无意义。winline() 与 line('.') 的意义不同显而易见，长文件经常滚动，窗口的第一行在不同时刻对应着文件的不同行。水平滚动条不如垂直滚动条用得多，但即使无水平滚动，wincol() 可能 也与 col('.') 不同。仍以上例汉字行，光标停在第四汉字上，wincol() 返回的是 7，因为前三汉字占 6 屏幕宽度，第四字从第 7 开始。

因为 Vim 可以分隔多个窗口，所以屏幕行列号 screenrow() screencol() 又与窗口 行列号 winline() wincol() 不同。不过屏幕行列号一般只用于测试。且直接在命令行手动输入 :echo screencol() 时，它始返回 1 ，因为执行命令时光标已经在命令行首列了。

- getpos() 获取光标或标记的位置信息
- setpos() 设定光标或标记的位置信息
- getcurpos() 获取当前光标的位置信息
- cursor() 放置当前光标

顾名思义，可能会觉得 getpos() 就是 line() 与 col() 的综合效果，但其实位置信息不仅是行列号。getpos() 的返回值是一个四元列表 [bufnr, line, col, off]， 其意义如下：

- bufnr 缓冲编号，0 表示当前缓冲，只有在取跨文件的全局标记，才需要返回其所在缓冲的编号，否则就是 0。
- line 行号，这就相当于 line() 函数了
- col 列号，这就相当于 col() 函数了
- off 偏移，只有在 &virtualedit 选项打开时才不是 0。比如 <Tab> 键可能 占多列，但在一般情况下移动光标时是直接跳过的，但在打开 &virtualedit 选项时 ，就可能移动到制表符中间某个位置了，这就是第四个返回值的意义。

setpos() 是 getpos() 的对应函数，它所接收的第二参数就是后者返回的四元列表。第一参数就是标记名 'x（注意含单引用，而非反引号） 或表示当前光标的 .。

getcurpos() 无参数，只返回当前光标的位置信息，基本与 getpos('.') 功能相同 ，不过返回值列表还多一个第五元素 curswant ，它表示当光标垂直移动（jk）时，它优先移动到的列号，因为当前列号在下一行或上一行未必是有效的，这时该移动到哪列呢，这第五个返回值就有效果了。

cursor() 用于放置当前光标，从语义上是 getcurpos() 函数的“反函数”，但是却不能将后者的返回参数传给前者。因为 getcurpos() 返回值是五元列表，而 curosr() 函数用不到其第一个返回值 bufnr，将第一个元素移除后的列表传给 cursor() 是可 行的。事实上，cursor() 还可以几个非列表的参数直接调用。如 cursor(line, col, off) ，或 coursor([line, col, off, curswant]) 当然，只有行列号是必须的。当 需要明确移动光标到某处时，直接调用 cursor(line, col) 是最方便的。当需要恢复 光标时，最好与 setpos() 联用，如：

```
: let save_cursor = getcurpos()
"
```

```
call setpos('.', save_cursor)
```

- byte2line() 文件的第几字节处于第几行
- line2byte() 第几行是从文件第几字节开始的

这两个函数将整个缓冲文件的字节索引与行号相互转换。注意包含换行符，换行符是一字节还是两字节则与文件格式有关。`line2byte(line("$") + 1)` 可获取缓冲的大小，其实比缓冲大小多 1，因为是文件最后一行的下一行的起始索引。除此之外，非法行号返回 `-1`。

- winheight() 返回指定编号窗口的高度，参数 0 表示当前窗口
- winwidth() 返回指定编号窗口的宽度
- winrestcmd() 返回一系列可恢复窗口大小的命令
- winsaveview() 保存当前窗口视图，返回一个字典
- winrestview() 由保存的字典恢复当前窗口视图

注意，`winrestcmd()` 只能恢复窗口大小，以字符串形式返回，将它用于 `:execute` 执行后才能恢复窗口大小。而 `winsaveview()` 与 `winrestview()` 能保存恢复比较完整的窗口信息。其参数字典保存哪些键名及释义请参阅相关文档。

- screenchar() 返回屏幕指定行列坐标的字符
- screenattr() 返回屏幕指定行列坐标的字符有关的特征属性

Vim 的屏幕不仅包括缓冲窗口，还有标签页行，状态栏，命令行，窗口分隔符等都占据一定屏幕坐标。不过这两个函数主要用于测试。

操作当前缓冲文本

然后是操作缓冲文件文本内容的函数，这是 Vim 作为文本编辑器的基础工作。

- getline() 从当前缓冲中获取一行文本字符串，或多行组成的列表
- setline() 从当前缓冲指定行开始替换文本行
- append() 从当前缓冲指定行下方始插入文本行
- getbufline() 从指定缓冲中获取文本行
- wordcount() 统计当前缓冲的字节、字符、单词，返回值是字典

如果 `getline()` 传入一个行地址参数，则返回一个字符串；如果传入两个起止行地址参数，则返回一个列表，每个元素为一行文本。行地址参数可以是数量或字符 `.` 表示 当前行，字符 `$` 表示最后一行。`setline()` 可传入一个行地址参数，以及一个字符串或字符串列表，用以替换指定行以及后续行。`appendline()` 用法与 `setline()` 一 样，不过是从指定行（下方）开始插入，并不会覆盖原有行。

`getbufline()` 与 `getline()` 类似，不过是取其他缓冲，所以要在第一个参数多传入一个缓冲编号或名字。另外，行地址参数不能用 `.` 点号表示当前行，因为在其他缓冲的当前行意义不明显（用户角度），而且返回值必定是列表，即使只有一个起始行地址参数，也是一个元素的列表。

- mode() 当前的编辑模式：普通、选择、命令行等
- visualmode() 上次使用的选择模式：字符、行、或列块选择

Vim 有很多种模式，在脚本中可用这两个函数获取模式信息，然后根据模式作不同的响应工作。一个非常有用的用途是用于状态栏定制中，否则触发该函数的时刻经常是命令行模式（通过命令行调用或加载脚本），或普通模式（映射中调用）。

- indent() 指定行的缩进空白列数
- cindent() 按 C 语法应该缩进的空白列数
- lispindent() 按 Lisp 语法应该缩进的空白的列数
- shiftwidth() 每层缩进的有效空白列数

这几个缩进函数其实都是只读函数，并不会改变缓冲内容（执行缩进操作的命令 =）。 **indent()** 是返回指定行（参数按 **getline()** 惯例）的当前实际缩进数，按缩进的空白数计，如果缩进字符是制表符，与相关的制表符宽度选项有关。而 **cindent()** 与 **listindent()** 是假设按 C 或 Lisp 语法规则缩进，该行应该缩进多少。需要根据这个返回结果调用其他命令或函数执行真正的修改操作。与缩进相关的选项有好几个，而 **shiftwidth()** 函数是综合这几个选项的设置，给出的当前缓冲实际生效的每级缩进数量。

- nextnonblank() 寻找下一行非空行
- prevnonblank() 寻找上一行非空行

这两个函数很简单，就是从参数指定的起始行地址查找非空行，如果起始行已经是非空行，直接返回该行地址。返回值是数字，失败时返回 0，因为行地址索引从 1 开始。作为 通用文本编辑器，Vim 假定文本文件用空行分隔段落。而且良好编程风格的大多数语言源文件，也是应该有空行分隔段落的，所以这两个函数有时挺实用。

- search() 搜索正则表达，返回行地址
- searchpos() 搜索正则表达式，返回行列号组成的二元列表
- searchpair() 按成对关键字搜索
- searchpairpos() 按成对关键字搜索
- searchdecl() 搜索一个变量的定义

这几个搜索函数可用于从脚本实现类似 / 的搜索命令，但有更灵活细致的控制。先看最基本的搜索函数的参数原型 **search(pattern, flag, stopline, timeout)**，只有第一个参数是必须的：

- **{pattern}** 就是 VimL 的正则表达式，在该函数中，一些影响搜索的选项如 **&ignorecase** ‘&magic’ 等将影响正则表达式的解析。
- **{flag}** 是一个字符串，每个字符表示不同的意义，一些冲突的标志不能并存：
- **b** 表示反向搜索，默认正向搜索；
- **c** 在光标处也能匹配成功；
- **e** 光标移动到匹配成功处的末尾，默认移动到匹配处的起始位置；
- **n** 即使匹配成功也不移动光标，但可利用函数的返回值，行地址；
- **p** 返回值不再是行地址，而是匹配成功的（或连接）子模式索引加 1；
- **s** 移动光标到匹配处前，将原位置保存在特殊标记 ' 中；
- **w** 搜索到文件末尾时，折回文件起始，与 **b** 并存时是到文件首折回；
- **W** 搜索到文件末尾或起始时不折回；
- **z** 从光标的列位置开始搜索，默认是从光标所在行首开始搜索。
- **{stopline}** 搜索从当前光标开始，可指定终止搜索的行。

- {timeout} 按毫秒数指定搜索的时间，搜索可能是个费时的操作，尤其是正则表达式写得复杂写得低效时，可指定时间，超时不再搜索。

所以这个函数有两个作用，一是返回值表示匹配的行地址，另一个副作用是会移动光标，除非指定 n 标志不移动光标。匹配失败时返回 0 ，当然也不会移动光标。

searchpos() 函数意义一样，只是返回多值（列表），除行号外，还返回列号，如果指定 p 标记，还返回所匹配的子模式索引（加 1）。这里的子模式是指由或操作 \| 连接的多个模式分支，匹配其中任一个都算匹配成功，但若需要知道匹配的是哪个分支，p 标记就有用了，注意需要被索引标记的子模式还得整个放在 \(\) 中。

searchpair() 成对搜索的意义类似在 VimL 脚本（或其他类似语法的语言，需加载自带的 matchit 插件）中在 if 关键字中按 % 命令，它会搜索配对的 endif 以及中间的 elseif。其参数就是在 search() 的基础上，将第一个正则表达式参数 {pattern} 换为三个正则表达式参数 ({start}, {middle}, {end}, ...)。并且可以在可选参数 {flag} 与 {stopline} 之间再加一个可选参数 {skip} ，其意义是表示如何忽略某些匹配，比如 elseif endif 在注释或字符串中应该是要忽略的。{skip} 是一个可执行字符串，当作表达式执行后返回非 0 就表示要忽略，执行时光标相当于已移动到匹配处。

searchpairpos() 的意义也类似，返回多值，即由行列号组成的列表而已。

searchdecl() 的作用与 gd 或 gD 普通命令类似，当然命令是取光标下的单词，函数需要将变量名字符串当作参数传入。

- getcharsearch() 获得字符搜索信息
- setcharsearch() 设定字符搜索信息

这两个函数是从 Vim8 版本新增的。字符搜索是指 f F t T 这几个命令用于实现行内搜索字符的，同时还有分号 ; 与逗号 , 按正反向重复上次字符搜索。如果要从脚本控制这种行为，可参考这两个函数。

修订窗口（quickfix）

很多命令会生成一个所谓的 quickfix 列表，这里将其译为修订。最早的应用来源是编译源代码给出的错误列表，每条项目会指出错误出现的文件、位置等，用于方便定位错误并修改。后来该概念扩展到其他许多命令，比如 grep 搜索，所以它就是一个有关定位的列表。该列表显示在单独的窗口中，就叫做修订窗口，可在该窗口预览各个“错误”信息，并像在普通窗口上移动，然后有方便的命令跳到相应位置外，并遍历整个列表。

据说最早的 Vim 版本并无此功能，只是一个插件功能，后来由于功能太过强大实用，就整合为 Vim 的内置功能了。而且还扩展出了局部修订列表的概念，即每个窗口都可以有自己的修订列表了。术语上，qflist 是全局的，locallist 是局部的。

- getqflist() 获得修订列表
- setqflist() 设置修订列表
- getlocallist() 获得局部修订列表
- setlocallist() 设置局部修订列表

getqflist() 返回的是字典列表，每个字典元素的键名解释请参考相应文档。setqflist() 接收这样的字典列表作为参数，并且有个可选的参数指出是添加到原修订

列表末尾还是覆盖原列表。后两个函数用法一样，不过在最前面多插入一个参数指出窗口编号（不是窗口ID）。

- taglist() 获得匹配的 tag 列表
- tagfiles() 获得 tag 文件列表

tag 文件是外部文件，记录着一些 tag （如变量名、函数名、类名等需要在大项目中检索与交叉引用的东西）的定义位置，该文件是由外部程序扫描（所有相关）源文件生成的，并遵循一定的格式。有了这样的文件，才能使用快捷键 C-] 与 :tag 命令。而 taglist() 是其函数形式，参数就是所要检过的 tag 名称，以正则表达式解析，要提供全名应自行加上 ^ 与 $ 界定。

Vim 使用的 tag 文件可用选项 &tags 设置，它是以逗号分隔的文件名字符串。函数 tagfiles() 返回的是当前缓冲实际所用的 tag 文件列表（VimL 列表类型）。

- complete() 设置补全列表
- complete_add() 向补全列表中增加条目
- complte_check() 检查是否终止补全
- pumvisible() 检查是否弹出补全窗口

插入模式下的补全是相对高级的话题。Vim 的默认模式是普通模式，定制插入模式本身就比较复杂。VimL 只提供了几个 api 函数。complete() 是简单地提供补全列表。complete_add() 与 complete_check() 只能用于自定义的补全函数（&compltefunc）中。

Vim 本身只是定位于通用文本编辑器，并非程序开发 IDE，但提供了这些基本接口，允许三方插件将其打造成的类似IDE的大多功能。尤其是 Vim8 版本新增的异步功能，能显著增加补全的性能与可用性。此不再详述，这些高级话题可能另辟章节讨论。

命令行信息

最后看几个有关命令行的函数。因为命令行也是可编辑区域，也是可以通过脚本访问的，不过一般只适于正在编辑命令行时使用，比如 :cmap 定义的映射等。

- getcmdline() 获得当前命令行
- getcmdpos() 获得光标在命令行的列位置
- setcmdpos() 设置光标在命令行的列位置
- getcmdtype() 获取命令行类型
- getcmdwintype() 获取命令行窗口类型

命令行类型比如通过 : / ? 进入的命令行都是属于不同的命令行类型。命令行窗口是通过特殊键在命令之上再打开的一个窗口，里面是命令行历史记录列表，可以方便选择某个历史命令或在彼基础上作小修改后再次执行。故 getcmdwintype() 只有在命令 行窗口时才有意义，其值与 getcmdtype() 相同。

- getreg() 获取某个寄存器的内容
- setreg() 设置某个寄存器的内容
- getregtype() 获取某个寄存器的类型

寄存器相当于 Vim 自己管理的剪贴板，允许用户自命名的寄存器有 26 个（即单字母表示），另外 Vim 还自动更新了许多以特殊符号表示的寄存器，各表示相应的特殊意义。寄存器

的内容可用 :registers 查看。这几个函数则用于脚本访问与控制寄存器。此外，对于常规字母命名的寄存器，以 @ 前缀的变量可直接表示该寄器（如 @a）。寄存器类型与选择类型（字符、行、列块）相烦，因为寄存器内容经常是选择后复制进去的。

用 Vim 编辑文本要善于利用命令行与寄存，这几个函数一般只在映射（调用）中比较有效果。

第六章 VimL 内建函数使用

6.3 操作外部系统资源

本节介绍的函数主要着眼于访问外部资源，比如最常用便是系统文件。

文件系统相关函数

- glob() 按文件通配符搜索文件
- globpath() 在系列目录中搜索文件
- findfile() 在搜索路径中查找文件
- finddir() 在搜索路径中查找目录

glob() 函数的作用，就相当于在 linux 终端命令 ls 所能列出的文件名。它可接收至多四个参数，只有第一个是必须的：

- {wildcard} 通配符文件名模式，非正则表达式；
- {nosuf} 让两个选项生效，&wildignore 可忽略某些文件，&suffixes 按文件名 后缀影响结果的排序；
- {list} 提供该参数则返回列表类型，否则是用换行符分隔的字符串；
- {alllinks} 一般情况下只会找出存在的文件，对于软链接文件，则其指向的文件有效才被包含在结果中，但若提供该参数，无效链接文件也接收。

一般第三个参数比较常用，即将结果按列表返回，以 glob(wild, 0, 1) 方式调用。

globpath() 用法是在 glob() 基础上，额外提供一个参数指定要哪些目录下搜索文件，必选参数，且插在第一个参数位置上。这是一个以逗号分隔的目录名列表字符串，如 &rtp 的表示法。例如 globpath(&rtp, 'readme.md') 就能搜索出所有运行时目录下的说明文档（目前许多插件安装习惯是安装在独立的运行时目录下，一般会有个 readme.md 说明文档）。

glob() 函数将返回所有匹配的文件名，但 findfile() 或 finddir() 只返回第一 个匹配的文件名，一个查找文件，一个查找目录，类似命令 :find 的作用。接收三个 参数，只有第一个必选：

- {name} 文件名，必须是全名，不是通配符；
- {path} 在这些目录下查找文件，也是逗号分隔的目录列表，省略的话用选项 &path 代替。所以实际所查到的文件名类似 {first-path}/{name}。
- {count} 指定返回第几个匹配的文件，而不是第一个，负数时返回所有匹配文件组成的列表类型变量。

Vim 的 :find 命令及 gf 命令使用 &path 选项值，这叫做搜索路径，这是搜索普通文件的；不同于 &rpt 运行时路径是搜索 vim 脚本的。搜索路径同时支持向下搜索与向上搜索的机制，在 {path} 参数或 &path 选项中使用特殊字符达成：

- 向下搜索：* 表示任意字符，** 表示任意子目录；
- 向上搜索：{one-path};{upto-path},{another-path} 即在一个路径（逗号分隔的）末尾再加一个分号，接一个相当基目录（{one-path}）更上层的目录（{upto-path}）就能从指定目录开始向上搜索，依次在其父目录搜索，直到终止目录 {upto-path}。 终止目录可省，但分号不可省，否则在该目录中就认为不需要支持向上搜索。建议不限定终止目录时写成 {base-path};/，或 {base-path};~，一直上溯到系统根目录或自己的家目录。
- 相对 Vim 当前路径写成单点 . ，相对当前正编辑的文件缓冲的路径写成 ./。

向上搜索机制，对于搜索工程项目文件很有用。比如当你正在编辑一个源代码文件，它一般被组织在各层子目录下，要找到项目文件就得使用向上机制了，例如 .git/ 目录或 tags 文件，都一般放在项目顶层目录中。

- resolve() 解析链接文件名
- simplify() 简化文件名路径
- pathshorten() 缩写文件名的中间路径
- fnamemodify() 文件名修饰

resolve() 是处理软链接文件（linux 系统）或快捷方式（MS-Windows 的 .lnk）的，将其转为实际指向的文件名。在其他系统同 simplify() 简化处理。文件名需要简化的一个例子是包含一系列的点号与双点号，如 ./dir/.././/file/，这可能是由其他函数拼接而来。simplify() 简化后不改变其意义，如上例简化结果为 ./file/。但是 pathshorten() 只是简单地将中间路径都缩写至首字母，显然是不保证其有效意义的。比如在默认的多标签页的名字，为节省屏幕空间就将当前编辑文件缩写目录名，~.vim/autoload/myfile.vim 将简写为 ~/.v/a/myfile.vim（如果觉得这比较丑，可寻插件定制标签页栏）。

文件名修饰是指如何从一个文件名中获取其目录、全路径名、后缀名等相关的名字字符串。函数 fnamemodify({fname}, {mods}) 的第二参数就叫做修饰符，修饰符以冒号开头带一个单字母表示不同意义，且可连续使用。主要的修饰符如：

- :p 文件全路径名
- :h 父目录名（文件名头部，去除路径分隔符最后一部分）
- :t 文件名尾部（一般是 :h 剩余部分，纯文件名）
- :e 文件名后缀
- :r 文件名主体（相对于 :e 而言，不包括后缀，但可能包含父目录）

注意 fnamemodify() 不处理特殊文件名变量，需用 expand() 先展开，不过后者也可以直接加修饰符后缀。如以下两个语句等效：

```
: echo fnamemodify(expand('%'), ':p:t')
: echo expand('%:p:t')
```

- executable() 检查是否可执行程序
- exepath() 可执行程序的全路径

- filereadable() 文件是否可读
- filewriteable() 文件是否可写
- getfperm() 获取文件权限（类 rwxrwxrwx 字符串）
- getftype() 获取文件类型
- isdirectory() 检测目录是否存在
- getfsize() 获取文件字节大小（目录返回 0）
- getftime() 获取文件的最后修改时间（整数，按秒计）

这几个函数用于检查指定文件的属性，其中 getftype() 返回的字符串主要有如：
* file 普通文件 * dir 目录 * link 软链接文件 * bdev cdev socket fifo
other 等

- getcwd() 获取当前工作路径
- haslocaldir() 检测当前窗口是否有局部当前路径（:lcd）

这两个函数都可选带两个参数，指定窗口编号与标签页编号，因为取当前窗口的当前路径
。Vim 启动时，从 shell 环境中继续当前路径，这是全局当前路径，可用 :cd
命令修 改。每个窗口可有自己的局部当前路径，这用 :lcd 修改。如果从未用过
:lcd ，窗 口的局部当前路径就与全局当前路径相同。新分裂的窗口继承原窗口的当前路径。:pwd
打印的是全局当前路径，因此有可能与 getcwd() 不同。

- mkdir() 创建新目录（类似 $mkdir）
- delete() 删除文件（类似 $rm）
- rename() 重命令文件（类似 $mv）
- readfile() 读文件至一个字符串列表
- writefile() 将字符串列表写入文件

这几个文件操作函数，除了 readfile() 返回列表外，其他函数在操作成功时返回
0 ，失败时返回非零错误码。其功能与相应的 linux 命令类似，不过将命令行参数改成函
数调用参数。如 mkdir('name', 'p') 类似 shell 命令 $mkdir -p name
可以自动 创建中间目录；delete() 删除非空目录时必须加参数 rf（谨慎）；rename()
重 命令文件可能覆盖已有文件无警告。当然这些操作也涉及系统权限。

读文件函数支持三个参数，readfile(fname, binary, max) ，后两个是可选的。默认
是按文本格式读入，主要会处理换行符。如果提供 {binary} 参数，按二进制格式读入
，虽然也会根据换行符分隔为列表元素，但元素中可能再保留回车符（dos
格式的文件） ，且最后可能多加一个空元素（若文件末尾是换行符）。第三个可选参数
{max} 可指 定只读入前几行，类似 linux 命令 $head -n ，但如果 {max}
参数是负数，则只读 入末尾几行，类似命令 $tail -n。

写文件函数要求两个参数，作为内容的字符串列表，以及文件名，还有个可选参数标记：
writefile(list, fname, flags)。标记 {flags} 若包含 b 则按二进制格式写入
，若包含 a 则添加到原文件末尾，否则是覆盖原文件。

一般情况下，Vim 是处理文件文本的，在使用这两个读写文件函数时，没必要指定
b 二进制格式。但是按二进制格式先 readfile() 再 writefile() 确实能达到复制文
件的作用。

调用外部系统命令

在 Vim 的命令行中，可用 :! 叹号开头，调用外部系统命令。而在 VimL 脚本中，相 应功能的函数是 system()。

- system() 执行系统命令，结果为字符串形式返回
- systemlist() 执行系统命令，结果以列表形式返回
- libcall() 调用外部库函数，结果返回字符串
- libcallnr() 调用外部库函数，结果返回数字

system(cmd, input) 将字符串 {cmd} 当作系统命令执行，返回字符串结果。如果 {cmd} 命令需要输入，则可提供可选参数 {input}，一般也是字符串，首先写入临时文件，再当作标准输入传给 {cmd}。如果 {input} 是字符串列表，则以二进制 b 方式调用 writefile() 写入临时文件。

{cmd} 命令字符串不支持管道。并且为了安全与正确性起见，最好调用 shellescape() 转义特殊字符。systemlist() 用法类似，只是返回结果是字符串列 表。

libcall() 类似于 call() 的基础用法，只是调用外库（.so 或 .dll）的函数，故需要库名、函数名与参数列表。当然不能随意调用外部库，只能调用专为扩展 vim 的 库，那才比较安全与实用。该函数将结果返回为字符串，另一个 libcallnr() 函数返 回的是数字结果。

- hostname() 获取 vim 所在运行的系统（计算机）名字
- getpid() 获取 vim 运行的进程号 PID
- tempname() 获取可用于临时文件的文件名

在实现比较复杂的功能时，可能需要用到临时文件，用 tempname() 获得一个可用的文件名（保证不重名）。也可以自己根据进程 PID 构建有规律的临时文件名。

日期时间函数

- localtime() 获取当前时间
- strftime() 格式化时间
- reltime() 获取相对时间
- reltimestr() 将相对时间转为字符串
- reltimefloat() 格式化相对时间转为浮点数

localtime() 用于获取当前的标准时间，即从 1970 年至今的秒数。将这样的时间转为可读模式，用 strftime(format, time) 函数，缺省 {time} 参数时，取当前时间，相当于先调用 localtime()。可用时间格式 {format} 与 C 语言的同名标准函数相同，如 strftime('%Y-%m-%d') 将返回类似 2017-11-11 的字符串。

reltime() 返回更精确的时间，具体格式与系统有关。无参数调用返回当前时间，一个参数 reltime(start) 返回从开始时刻（{start}也应该是由该函数返回的）到现在所经过的时间，两个参数 reltime(start, end) 返回两个时刻之间的时间。用 reltimestr() 将这样的时间转为字符串表示，reltimefloat() 转为浮点数表示，因为字符串表示法也正像个浮点数（即秒数加小数点加毫秒数）。因其精确到毫秒，可用来计算命令或函数执行的时间。

用户交互函数

- input() 获得用户从命令行输入的一行文本
- inputsave() 保存用户输入序列
- inputrestore() 恢复用户输入序列
- inputsecret() 按密文输入
- intputdialog() 从对话框中输入一行文本

在 VimL 脚本中与用户交互的最常用的函数是 input(, ,)。提示字符串参数必须给，可以是空字符串，也可以用 \n 表示多行提示。后面两个参数可选。Vim 首先在命令打印提示字符串，等待用户输入一行文本，按回车返回用户刚输入的这行文本。如果直接回车没任何输入，则返回传给函数的默认值（或空字符串）。当用户输入时，相当于编辑命令行，所以为便于用户输入，可提供补全方法，类似自定义命令那般。而且用户的输入也有独立的命令行历史记录。

显然，input() 函数不宜用于启动配置 vimrc 中。此外，也要避免用于映射中，因为映射的后续键相当于用户输入，会当作 input() 的回应输入。如果一定要用于映射中，请在调用 input() 前后分别调用 inputsave() 与 inputrestore()。

inputsecret() 用法一样，只是用户输入的文本不直接显示在屏幕命令行中，以星号 * 代替。此外也不支持补全，不放入历史记录中，因为这主要用于提示输入密码。

在 GUI 版本中，inputdialog() 可弹出对话框，让用户从对话框中输入，否则类似 input() 函数。

- inputlist() 让用户从一个列表中选择一项
- confirm() 也是让用户从列表中选择一项

inputlist() 接收一个字符串列表参数，Vim 将每个元素一行显示在命令行上方的消息区，然后提示用户输入一个数字选择一项（GUI 版本可用鼠标）。注意按列表索引惯例，0 表示选择第一项。为弥补这个反人类设计，这有个技巧：将提示文本写在列表的第一项，后续有效选项字符串也以索引 1. 2. 之类的开始，让用户能直观地选择数字。

让用户做选择还有另一个函数 confirm()，它可用于 GUI 版本，也可用于终端版本。它可接收四个参数，confirm( )，一般只用到 前两个。与 inputlist() 不同的是，提示文本为独立参数，且选项列表是字符串，用 回车分隔每一选项，且第一项是 1 。在每一项的字符串中，可以将 & 加在某个字符 之前，则按该字符时直接选择了项目（选择快捷键），且不像 inputlist() 那样会将 所按键显示在命令行中（因为其实这是为GUI版本设计的），也不需要多按回车确认，就是快捷键直接选择。当然函数返回的仍是选项索引，并非快捷键字符。可选的默认选项参数也应该是数字索引，不提供时默认 0 ，算是无效选项。

- getchar() 获取用户按下的下一个键
- getcharmod() 获取用户按键时的修饰键
- feedkeys() 将一个字符串放入 vim 待响应的按序序列

getchar() 用于获取用户（或输入流）的下一个键。不同于 input() 进入命令行等待交互，而是默默地等待获取下一输入键。相当细节很多，用到时请参考文档。因为 vim 本身的总体工作（消息）循环，就是等待用户按键，然后作出不同响应。

getcharmod() 用于获取修饰键（收到上个键时同时按下的修饰键），如 shift = 2 、ctrol = 4 、alt = 8 等。将可用修改键用二进制编码，返回一个数字就能表示哪

些修饰键被按下了。

feedkeys() 的用途就比较诡秘了。它把一个字符串放回输入流中，当作是用户的按键输入序列。特殊按键用 **"\< >"** 表示。默认情况下，放回的这些字符键是可再被重映射的，然而也有一些可选参数控制细节。

- browse() 打开浏览文件对话框
- browsedir() 打开目录选择对话框

这两个函数只能用于 GUI 版本，弹出标准对话框，让你选择一个文件或目录，返回所选择的文件路径名。可以传入参数指定对话框标题及初始浏览目录。

- getfontname() 获取当前所用的字体
- getwinposx() 获取 gVim 窗口的坐标
- getwinposy() 获取 gVim 窗口的坐标

这几个函数只能用于 GUI 版本，检索 GUI 才用得到的信息。

异步通讯函数

自 Vim8 版本引入了一些全新的特性：任务（job）、定时器（timer）、通道（channel），这都涉及异步编程，主要通过回调函数实现功能。为此也提供了一系列相关的内建 api 函数。不过本章不想罗列这些函数，毕竟需要理解相应的功能才有理解函数用法的意义。留待后续章节专门讨论吧。

第六章 VimL 内建函数使用

6.4 其他实用函数

在本章的最末，再介绍一些不太分类，或不常用（但不甚复杂）的函数。须要再次强调的是，VimL 的函数，是为了访问或（与）控制相应的 Vim 功能而设的。必须理解相应的功能才能用好相应的函数。Vim 提供的功能很多，对于个体用户而言，可能对某些功能并不关注或并不感兴趣。

因此，本章罗列介绍这些函数，无法求细致，只为说明 VimL 有这么一种类的函数可用。当你真正需要用到时，再回去查手册。任一编程语言的 api 函数，只能通过手册学习，通过实践提高。而这无法从任一书籍教程中学得，书籍只能是引入门，帮用户建立个相关概念而已。

特性检测函数拾遗

- has() 当前 Vim 版本是否编译进某个功能，似 **:verstion** 功能
- exists() 检查是否存在某个变量、命令、函数等对象
- type() 返回变量类型

其中 **exists()** 函数功能强大，用参数字符串前缀来表示哪类对象，主要有以下几种：
* 选项：**&option_name** 只判断选项存在，**+option_name** 判断选项是否有效
* 环境变量：**$ENV_NAME** * 函数：***function_name** * 命令：**:command_name**
对于函数与命令，都可检查内建的或自定义的 * 变量：**variable_name** 即没有特殊前缀时检查变量是否存在
* 自动事件：**#event #group** 等

- hasmapto() 检查某个命令（键序列）是否有映射（{rhs}）
- mapcheck() 检查某个按键序列是否有被映射（{lhs}）
- maparg() 获得某个被映射的键序列实际映射键序
- wildmenumod() 在命令行映射中检查是否出现补全模式

这几个检查映射状态的函数，常用于设计可定制插件的映射，避免重复、覆盖或冗余的映射。maparg() 与 mapcheck() 的主参数都是映射的 {lhs}，前者要求精确匹配，后者只需匹配前缀，返回映射到的 {rhs}。hasmapto() 是反向检查是否有任意的 {lhs} 映射到参数指定的 {rhs}。

类型文件语法相关

文件类型是 Vim 的重要概念，简单理解的话，不同的文件名后缀往往代表不同类型的文件（&filetype）。比如不同编程语言的源代码文件。不过 Vim 只是编辑器，它并不能理解（编译或解释）任一种编程语言（VimL 除外）。所以 Vim 语境下的“语法”，本质上只是“词法”，旨在说明可以如果对文件的不同部分进行不同的高亮着色，不过习惯上仍称为语法着色。

这主要分两步实现。首先是定义高亮组（highlight group），它说明“如何高亮”，描叙了该高亮下的颜色、字体等信息。然后是定义语法项（syntax），它说明高亮什么，主要是基于正则表达式，将不同匹配（match）部分应用不同的高亮组。这样就有可能将一个缓冲文件在窗口中以五颜六色的方式呈现出来。

一般不同的文件类型有相应的语法文件，文件中主要用 :highlight 与 :syntax 命令定义了各种高亮组与语法项。这里要介绍的 VimL 内置函数，可以检索当前缓冲中实际生效的语法信息，以及临时增加修改部分高亮方式。

- matchadd() 增加一种匹配应用某种高亮组
- matchaddpos() 在特定（行列）位置上匹配应用高亮
- matchdelete() 删除一处匹配的高亮
- clearmatches() 清除所有匹配高亮
- matcharg() 获取 :match 命令的参数信息
- getmatches() 获取所有自定义匹配高亮
- setmatches() 恢复一组自定义匹配高亮

这几个函数用于手动控制一些文本的高亮方式（相对于语法文件的自动加载）。虽然这种方式似乎比较原始，但有助于理解 Vim 语法高亮的处理机制，并且在临时微调语法高亮或处理一些简单非常规文件类型时也有奇效。

首先要了解 :match 命令，它于之前介绍的用于匹配字符串的 match() 函数是不同 的功能。:match {group} /{patter}/ 的意思相当于临时定义一种语法匹配，将匹配正则表达式的文本应用指定的高亮组。你可以初步地认为每种文件类型的特定语法文件中都正式地定义了很多种类似的语法匹配。不过 :match 的临时定义只能定义三种不同的匹配，另外两个命令是 :2match 与 :3match。实际上它只是 :match 命令的数字参数，默认是 :1match 而已。限制三种可能是出于性能、管理与实用的综合考虑。

然后 matchadd() 是 :match 命令的函数方式。所不同的是它不限于只定义三种匹配，可以调用这个函数定义许多匹配，并且返回不同的 ID 用以表示所定义的不同匹配。当然 1-3 这前三个 ID 被保留给 :match 命令使用。

matchaddpos() 的功能类似，不过它不是通过正则表式式来定义匹配，而是准确地给出一组位置信息，说明在哪些行（列）上要高亮。因为基于正则的语法高亮是比较低效的，（在一些旧机器上打开大文件时若发现卡，可尝试关闭语法着色），按位置高亮就不那么耗性能了。

matchdelete() 用于删除自定义匹配，参数就是 matchadd() 返回的匹配ID（或者 1-3 代表通过 :match 命令定义的）。clearmatches() 则清除所有自定义匹配，不需要参数。

matcharg() 就是用于查看之前的自定义匹配，接收匹配ID为参数，返回其定义的信息。getmatches() 用于获得所有自定义匹配的详细信息，返回的是字典列表，它可传给 setmatches() 恢复自定义匹配。

- hlexists() 由高亮组名判断其是否存在
- hlID() 由高亮组名返回其数字ID
- synID() 返回当前缓冲指定行列位置处所使用的语法项ID
- synIDattr() 由语法项ID返回可读的属性信息
- synIDtrans() 返回一个语法项ID实际所链接的语法项ID
- synstack() 返回当前缓冲指定位置处所有的语法项ID堆栈
- synconcealed() 返回当前缓冲指定位置处的隐藏语法信息

在 Vim 内部，为每个高亮组与语法项都分配了 ID，其中高亮组有名字，而语法项是个更虚拟的概念，没有名字，只能用 ID 表示。在当前缓冲的某部分文本（以行列号为参数）使用了什么语法高亮，可用 synID() 获得，据此可进一步由 synIDattr() 获得该语 法高亮的详情属性。

在语法文件中，普遍使用 :highlight link 命令链接高亮组。因为 Vim 预设了大量通 用的高亮组名字，但允许用户在为不同类型文件的语法中使用更有意义的高亮组名，为避免重复定义高亮属性，就可以将新高亮组链接到既有高亮组。synIDtrans() 函数就是 用于获得某个语法项ID实际链接的语法项ID。

Vim 的语法定义其实不是简单的正则匹配，还有更复杂的区域（region）与嵌套规则。于是对于一部分文本（缓冲的行列位置）而言，它可能不只应用了一个语法项高亮，而是由内向外形成了语法高亮栈。synstack() 就是获取这样的栈的函数，它返回一个列表，最后一个元素其实就是 synID()。

- foldclosed() 检查当前缓冲指定行是否被折叠，返回折叠区的第一行
- foldclosedend() 同上，返回折叠区的最后一行
- foldlevel() 返回当前缓冲指定行应该折叠的最深层次，不要求已折叠
- foldtext() 默认的折叠行显示文本计算函数
- foldtextresult() 当前缓冲指定行如果折叠，折叠行应该显示的最终文本

折叠从某种意义来说，也属于语法规则的范畴，只是它不是关于如何着色的，而是定义文本层次的。折叠方法用很多种，可由选项 &foldmethod 指定，其中一种就是 syntax 由语法定义决定折叠层次。有很多普通命令（z 开头的系列）处理折叠。这里的几个函数只是访问折叠信息。

如果当前缓冲的某行已被折叠中，函数 foldcolsed() 与 foldclosedend() 分别返 回整个折叠区的首行与尾行；若未折叠，返回 -1。据此可知缓冲的实际文本与窗口显示的文本行的差异。foldlevel() 返回折叠层级。任何折叠方法最终都由每行的折叠层

级决定折叠行为。其中有种自定义折叠函数（表达式），就由用户自己控制、计算返回每行的折叠层级。其他折叠方法 Vim 会自动计算折叠层级。

折叠后，会在折叠首行显示一行特征文本，这行文本的计算方法由选项 `&foldtext` 配置指定。其默认值就是 `foldtext()`，该函数也只能在计算 `&foldtext` 时调用。某行折叠后实际将显示的特征文本，则由 `foldtextresult()` 给出。

测试函数

当程序或脚本变得复杂起来，单元测试就可能很有必要了。从 Vim8 版本始，也提供了许多函数方便用户写单元测试。

- `assert_equal()` 断言两个值相等，包括变量类型相同
- `assert_notequal()` 断言不相等
- `assert_inrange()` 断言一个值处在某个范围
- `assert_match()` 断言匹配正则表达式
- `assert_notmatch()` 断言不匹配正则表达式
- `assert_false()` 断言逻辑假，或数字0
- `assert_true()` 断言逻辑真，或非0数字
- `assert_exception()` 断言会抛出异常
- `assert_fails()` 断言执行一个命令会失败

这些断言函数用法类似，一般是接收预期值与实际值，如果不满足断言条件，就添加一条消息至内置变量 `v:errors` 列表中，其中消息字符串能传入可选参数定制。这些函数本身不会产生错误输出或中断运行，只是先将错误信息暂存至 `v:errors` 中。所以在做单元测试中，应该先将 `v:errors` 列表置空，调用一系列断言函数，最后检查该列表保存了哪些错误消息，如果一切正常，该列表应该是空的。

第七章 VimL 面向对象编程

面向对象是一种编程思想，并不特指某种编程语言。所以不必惊讶用 VimL 也能以面向对象的方式来写代码。本章先简单介绍一下面向对象的编程思想，再探讨如何利用 VimL 现有的特性实现面向对象编程。最后应由用户自行决定是否有必要使用面向对象的风格来写 VimL 脚本。

7.1 面向对象的简介

在前文中用了比较多的篇幅来介绍函数。如果主要以函数作为基本单元来组织程序（脚本）代码，函数间的相互调用通过参数传递数据，这种方式或可称之为面向过程的编程。大部分简单的 VimL 脚本都可以通过这种方式实现。单元函数的定义与复用也算简洁。

但是，如果有更大的野心，想用 VimL 实现较为复杂的功能时，只采用以上基于函数的面向过程编程方式，可能会遇到一些闹心的事情。比如函数参数过多，需要特别小心各参数的意义与次序，或许还可能不可避免要定义相当多的全局变量。当然，这可能并不至于影响程序的功能实现，主要还是对程序员维护代码造成困扰，增加程序维护与复用的困难。

这时，就可考虑面向对象的编程方式。其核心思想是数据与函数的整合与统一，以更接近人的思维方式去写代码与管理代码。

面向对象的基本特征

按一些资料的说法，面向对象包含以下四个基本特征：

- 抽象
- 封装
- 继承
- 多态

严格说来，任何编程都应该从抽象开始。分析现实需求问题的主要关系，归纳出功能单元组成部分。按面向过程编程方式设计函数时，同样也需求程序员的抽象能力。所以也有些教程资料说面向对象的基本特征是后面这三个：封装、继承与多态。这又涉及面向对象的另一个关键概念，类。

类就是将现实诸问题抽象后的封装结果。它包括数据以及操作这些数据的方法，从概念及表现上将这两部分放在一起视为一个整体，就称之为封装。类往往对应着现实世界的某种类型的实体或动作。我们一般只用关注某类物事的表面接口，而不必关心其内部构造细节。反映到程序上，类的封装就是为了隐藏实现，简化用法，一般用户只要理解某个类是什么或像什么，以及能做什么，则不用深究怎么做。

所以在程序中，类不外是一种可自定义的复杂类型。与之相对应的简单类型就是如数字、字符串这种在大多数语言都内置支持的。简单类型除了可以用值表示一种意义外，还支持特定的操作，如数字的加减乘除，字符串的联连、分割、匹配等。类也一样，它用于表示值的就是被封装（可能多个）数据，也常被称为成员属性，它所支持的操作方法也叫成员函数。而对象与类的关系，也正如变量与类型的关系。对象是属于某个类的，有时称其为实例变量。

有些语言的面向对象还对类的封装进行了严格的控制，比如从外部访问对象只能通过类提供的所谓公有方法（属性），而另外一些私有方法（属性）只能在类内部的实现中使用。

继承是为了拓展封装之后的类代码的复用，将一个类的功能当作一个整体复用到另一个相关的类中。这也是对现实世界中具有某种从属关系的事件的一种抽象。在被继承与继承的两端，一般称之为基类与派生类，或通俗点叫父类与子类。子类继承了父类的大部分属性与方法（具体的语言或由于访问权限另有细节控制），因而可以像操作父类一样操作子类。

多态是为了进一步完善继承的用途而伴生的一个功能实现概念，使得在一簇继承体系中，诸派生类各具个性的同时，也保留共性的方法访问接口。即向许多对象发送相同的消息使其执行某个操作，各对象能依据其类型作不同的响应（功能实现）。

面向对象示例分析

先举个概念上的例子。就比如数字，仅管很多语言把数字当作简单的内置类型来处理，却也不妨用类与对象的角度来思考这个已经被数学抽象过的概念。

我们知道数字有很多种：整数、实数、有理数、复数等。每种数都可以抽象为一个类，还可以在这之上再抽象出一种虚拟的“数字”类，当作这些数类的统一基类。这些类簇之间就形成了一个继承体系。凡是数字都有一些通用方法，比如说加法操作。用户使用时，只需对一个数字调用加法操作，而不必关心其是哪类数，每类数会按它自己的方式相加（如有理数的相加与复数的相加就有显著不同）。这就是使用上的多态意义。整数一般可以用少

数几个字节（四或八字节）来表示，但如果有时要用到非常大的整数，可能需要单独再定义一个无限制的大整数类。但对一般用户来说，也不必关心大整数在底层如何表示，只需按普通小整数一样使用即可，这就是封装的便利性。

再举个切近 Vim 主题的例子。Vim 是文本编辑器，它主要处理的业务就是纯文本文件。那么就不妨将文本文件抽象为一个类。其实从操作系统的角度讲，文件包括文本文件与二进制文件，若按“一切皆文件”的 linux 思想，其他许多设备也算文件。然而，以 Vim 的 功能目的而言，可不必关心这些可扩大化的概念，就从它能处理的文本文件作为抽象的开始吧。

Vim 将它能编辑的文件分为许多文件类型，典型的就如各种编程语言的源代码文件。于是每种文件类型都可视为文本文件这个“基类”的“派生类”。然后，Vim 所关注的只是编辑源码，并不能编译源码，它只能处理表面上的语法（或文法）用于着色、缩进、折叠等美化显示或格式化的工作。所以不妨再把一些“语法类似”的语言再归为同一类，比如 C/C++、 java、C#、javascript 等（都以大括号作为层次定界符），就可以在其上再抽象一个 C-Family 的基类，它处于最基本的文本文件之下，而在各具体的文件类型之上。显然，类的抽象与设计，是与特定的功能目标有关的。若在其他场合，将 C++、java、javascript 等傻傻分不清混为一谈就可能不适合了。

若再继续分析，C 语言与 C++ 语言还算是不同的语言，总有些细节差异需要注意，尤其是人为规定的源码编程风格问题。至于是否真要再细分为两个类，那得看需求的权衡了。另外，C/C++ 语言还有个特别的东西，它要分为头文件与实现文件。这也得看需求是否要再划分为两个类设计。如果在编写 C/C++ 代码时需要经常在头文件声明与实现文件的实现时来回跳转，甚至想保持实现文件的顺序与头文件声明一致以便于对照阅读，那么再继承两个类分别设计或许是有意义的呢。

对所有这些语言源码文件，Vim 都提供了一个缩进重格式化的功能（即 = 命令）。只 要为每个类实现重缩进的操作（实际上利用了继承后，也只要在那些有差异需求文件类型上额外处理），就可以让 Vim 用统一的一键命令完成这个工作了。这就相当于多态带来的便利。

当然了，以上的举例，只是概念上的虚拟示例。Vim 编程器本身是用 C 语言写的，并没有用到面向对象的方式，因而也不会为文件类型设计什么类。而且既然它主要是为处理文本，VimL 也只要处理简单的整数与实数（浮点数）即可，不会去设计其他复杂的数字类。这主要是说明如何采用面向对象的思想分析问题，提供一种思路与角度，顺便结合示例再说明下面向对象的几个特征。

面向对象的优劣提示

上文介绍了面向对象的特征，由此带来代码易维护易管理的优点。同时上面的例子也说明面向对象并不是必要，不用面向对象也能做出很好的应用产品。

其实，面向对象主要不是针对程序，而是针对程序员而言的。如果简单功能，单人维护，尤其是一次性功能，基本就不必涉及面向对象，因为要实现对象的封装会增加许多复杂代码。面向对象适合的是复杂需求，尤其涉及多人协作或需要长期维护的项目。此外，在实际使用面向对象编程时，也要注意避免类的过度设计，增加不必要复杂度。

本章剩下的内容旨在探讨如何使用 VimL 实现基本的面向对象。从学习的角度而言，也可据此更深入地了解 VimL 的语言特性。至于在实践中，开发什么样的 Vim

功能插件值得 使用面向对象编程，那就看个人的需求分析与习惯喜好了。

第七章 VimL 面向对象编程

7.2 字典即对象

字典是 VimL 中最复杂全能的数据结构，基于字典，几乎就能实现面向对象风格的编程。在本章中，我们提到 VimL 中的一个对象时，其实就是指一个字典结构变量。

按对象属性方式访问字典键

首先要了解的是一个语法糖。一般来说，访问字典某一元素的索引方法与列表是类似的，用中括号 [] 表示。只不过列表是用整数索引，字典是用字符串（称为字典的键）索引。例如：

```
: echo aList[0]
: echo aDict['str']
```

（这里假设 aList 与 aDict 分别是已经定义的列表与字典变量）

如果字典的键是常量字符串，则在中括号中还得加引号，这写起来略麻烦。所以如果字典键是简单字符串，则可以不用中括号与引号，而只用一个点号代替。例如：

```
: echo aDict.str
```

这就很像常规面向对象语言中访问对象属性的语法了。所谓简单字符串，就是指可作为标识符的字符串（比如变量名）。例如：

```
: let aDict = {}
: let aDict['*any_key*'] = 1
: let aDict._plain_key_ = 1
: let aDict.*any_key* = 0     |"
```

然后要提醒的是，字典键索引也可用字符串变量，那就不能在中括号内用引号了。当要索引的键是变量时，中括号索引语法是正统，用点号索引则不能达到类似效果，因为点号索引只是常量键的语法糖。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    |"      aDict['str_var']

: echo aDict.{str_var}   |"
: let prefix = 'some_'
: echo aDict.{prefix}key   |"
: let prefix = 'a'
: echo {prefix}Dict.some_key |" --> some value
: let midfix = '_'
: echo aDict.some{midfix}key   |"
```

上例的后半部分还演示了 VimL 的另一个比较隐晦（但或许有时算灵活有用）的语法，就是可以用大括号 {} 括住字符串变量内插拼接变量名，这可以达到使用动态变量名的效果。但是，这种拼接语法也只能用法普通变量名，并不能用于字典的键名。键名毕竟与变量名不是同种概念。（关于大括号内插变量名的语法，请参阅 :h curly-braces-names）

总之，将字典当作对象来使用时，建议先创建一个空字典，再用点索引语法逐个添加属性。可以在用到时动态添加属性，不过从设计清晰的角度看，尽可能集中地在一开始初始化主要的属性，通过赋初值，还可揭示各属性应保存的值类型。例如，我们创建如下一个对象：

```
: let object = {}
: let object.name = 'bob'
: let object.desc = 'a sample object'
: let object.value = 0
: let object.data = {}
```

从上例中便可望文生义，知道 object 有几个属性，其中 name 与 desc 是字符串类型，value 是一个数字，可能用于保存一个特征值，其他一些复杂数据就暂存 data 属性中吧，这是另一个字典，或也可称之为成员对象。当然了，VimL 是动态类型的语言 ，在运行中可以改变保存在这些属性中的值的类型，然而为了对程序员友好，避免这样做。

字典键中保存对象方法

如果在字典中只保存数据，那并不是很有趣。关键是在字典中也能保存函数，实际保存的是函数引用，因为函数引用才是变量，才能保存在字典的键中，但在用户层面，函数引用与函数的使用方式几乎一样。

保存在同一个字典内的数据与函数也不是孤立的，而应是有所联系。在函数内可以使用在同一个字典中的数据。用形象的话，就是保存在字典内的函数可以操作字典本身。这就是面向对象的封装特性。字典键中的函数引用，就是该对象的方法。

在 VimL 中，定义对象的方法也有专门的语法（糖），例如：

```
function! object.Hello() abort " dict
    echo 'Hello ' . self.name
endfunction

: call object.Hello()    |" --> Hello bob
```

在上例中，object 就是已经定义的字典对象，这段代码为该对象定义了一个名为 Hello 的方法，也即属性键，保存的是一个匿名函数的引用；在该方法函数体内，关键字 self 代表着调用时（不是定义时）的对象本身。然后就可以直接调用 :call object.Hello() 了，在执行该调用语句时，self 就是 object 。

按这种语法定义对象方法时，可以像定义其他函数一样附加函数属性，其中 dict 属性 是可选的，即使不指定该属性，也隐含了该属性。之所以说这也像是一个“语法糖”，是因为这个示例相当于以下写法：

```
function! DF_object_Hello() abort dict
```

```
    echo 'Hello' . self.name
endfunction
let object.Hello = function('DF_object_Hello')
```

这里函数定义的 dict 属性不能省略，否则在函数体内不能用 self。不过这仍是伪语法糖，因为这两者并不完全等效，后者还新增了一个全局函数，污染了函数命名空间。而上节介绍的点索引属性，object.name 才是与 object['name'] 完全等效的真语法糖。

从 VimL 的语法上讲，在字典键中保存的函数引用，可以是相关的或无关的函数。但从面向对象设计的角度看，若往对象中添加并无关联的函数，就很匪夷所思了。例如下面这个方法：

```
function! object.HellowWorld() abort dict
    echo 'Hello World'
endfunction
```

在这个方法内并未用到 self，也就说不需要对象数据的支持，那强行放在对象中就很没必要了。要实现这个功能，直接定义一个名为 HelloWorld 的全局函数（或脚本局部函数）就可以了。

然而，一个函数方法是否与对象有关，这是一种抽象分析的判断，并非是从语法上函数体内有无用到 self 来判断。假如上面这个 object 的用于打招呼的 Hello() 方法另增需求，除了打印自身的名字外，还想附加一些语气符号。我们也将这个附加的需求抽象为函数，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!!!
```

这里的 EndHi() 方法也不需要用到 self，不过从它的意途上似乎与对象相关，所以也存在字典对象的键中，也未尝不可。

在一些面向对象的语言中（如 C++），这种用不到对象数据的方法可设计为静态方法，它是属于类的方法，而不是对象的方法。那么，在 VimL 中，可以如何理解类与对象的区别呢？

复制类字典为对象实例

事实上，VimL 只提供了字典这种数据结构，并没有什么特殊的对象。所以类与对象都只能由字典来表示，从语法上无从分辨类与对象，这只能是由人（程序员）来管理。通过某种设计约定把某个字典当作类使用，而把另一些字典当作对象来使用。

这首先是要理解类与对象的关系。类就是一种类型，描叙某类事物所具有的数据属性与操

作方法。对象是某类事物的具体实例，它实体拥有特定的数据，且能对其进行特定的操作。从代码上看，类是对象的模板，通过这个模板可以创建许多相似的对象。

在上节的示例中，我们只创建了一个对象，名为 object。可以调用其 Hello() 方法，效果是根据其名字打印一条欢迎致辞。如果要表达另一个对象，一个笨办法是修改其属性的值。例如：

```
: call object.Hello()    |" --> Hello bob!!!
: let object.name = 'ann'
: call object.Hello()    |" --> Hello ann!!!
```

但是，这仍然只实际存在了一个对象。如果在程序中要求同时存在两个相似的对象，那该如何？也很容易想到，只要克隆一个对象，再修改有差异的数据即可。当然了，你不用在源代码上复制粘贴一遍对 object 的定义，只要调用内置函数 copy()。因为有关该对象的所有东西都已经在 object 中了，在 VimL 看来它就是一个字典变量而已。如：

```
: let another_object = copy(object)    |"  deepcopy(object)
: let another_object.name = 'ann'
: call another_object.Hello()    |" --> Hello ann!!!
```

不过要注意的是，由于当初在定义 object 时，预设了一个字典成员属性 data。如果用 copy(object) 浅拷贝，则新对象 another_object 与原对象 object 将共享一份 data 数据，即如果改变了一个对象的 data 属性，另一个对象的 data 属性也将改变。如果设计需求要求它们相互独立，则应该用 deepcopy(object) 深拷贝方法。

以上用法可行，但不尽合理。因为在实际程序运行中，object 的状态经常变化，在一个时刻由 object 复制出来的对象与另一个时刻复制的结果不尽相同，且不可预期。那么就换一种思路。可以先定义一个特殊的字典对象，其主要作用只是用来“生孩子”，克隆出其他对象。即它只预定义（设计）必要的属性名称，及提供通用的初值。当在程序中实际有使用对象的需求时，再复制它创建一个新对象。于是，这个特殊的字典，就可充当类的作用。类也是一个对象，不妨称之为类对象。

于是，可修改上节的代码如下：

```
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!!!
```

这里，先定义了一个类对象，取名为 class。其数据属性与方法函数定义与上节的 object 几乎一样，不过是换了字典变量名而已。另外，既然 class 字典是被设计为当作类的，不是实际的对象实例，那它的 name 属性最好留空。当然了，取名为 noname 之类的特殊字符串也许也可以，不过用空字符串当作初值足够了，且节省空间。然后，使用这个类的代码就简单了，由 deepcopy() 创建新对象，然后通过对象访问属性，调用方法。

这就是 VimL 所能提供的面对象支持，用很简单的模型也几乎能模拟类与对象的行为。当然了，它并不十分完美，毕竟 VimL 设计之初就没打算（似乎也没必要）设计为面向对象的语言。如果在命令输入以下命令查看这几个字典对象的内部结构：

```
: echo class
: echo obj1
: echo obj2
```

可以发现，对象 obj1 obj2 与类 class 具有完全一样的键数量（当然了，因为是用 copy 复制的呀）。VimL 本身就没有在对象 obj1 与类 class 之间建立任何联系，是我们（程序员）自己用全复制的方式使每个对象获得类中的属性与方法。每个对象应该有自己独立的数据，这很容易理解与接受。但是，每个对象都仍然保存着应该是通用的方法，这似乎就有些浪费了。幸好，这只是保存函数引用，不管方法函数定义得多复杂，每个函数引用都固定在占用很少的空间。不过，蚊子再小也是肉，如果一个类中定义了很多方法，然后要创建（复制）很多对象，那这些函数引用的冗余也是很可观的纠结了。

另外，直接裸用 copy() 或 deepcopy() 创建新对象，似乎还是太粗糙了。如果数据属性较多，还得逐一赋上自己的值，这写起来就比较麻烦。因此，可以再提炼一下，封装一个创建新对象的方法，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!!!
```

不过，仍然有上面提及的小问题，class 中的 new() 方法也会被复制到每个新建的对象如 obj3 与 obj4 中。若说在类中提供一个 new() 方法很有意义，那在对象实例中也混入 new() 方法就颇有点奇葩了，只能选择性忽略，不用它就假装它不存在。当然了，如果只是想封装“创建对象”这个功能，也可用其他方式回避，这在后文再叙。

这里要再提醒一点是，由于 new() 方法是后来添加进去 class 类对象中的。在这之

前创建的 obj1 obj2 对象是基于原来的类定义复制的，所以它并不会有 new() 方法，在这之后创建的 obj3 obj4 才有该方法。如果在之后给类对象添加新的属性或（实用）方法，也将呈现这种行为，原来的旧对象实例并不能自动获得新属性或方法。有时固然可以有意利用这种动态修改类定义的灵活特性，但更多的时候应该是注意避免这种无意的陷阱。这也再次说明了，VimL 语法本身并不能保证对象与类之间的任何联系，其间的联系都是“人为的假想”，或者说是程序员的设计。

复制字典也是继承

上文已经讲了，通过在字典中同时保存数据与函数（引用），就基本能实现（模拟）面向对象的封装特征。然后，面向对象的另一个重要特征，继承，该如何实现呢？其实一句话点破也很简单，也就是用 copy() 复制，复制，再复制。

接上节的例子，我们打算从 class 类中继承两个子类 CSubA 与 CSubB ，示例代码如下：

```
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###
```

在这两个子类中，我们只重写覆盖了 EndHi() 方法，让每个子类使用不同的语气符号后缀。而基类 class 中的 new() 方法与 Hello() 方法，自动被继承（就是复制啦）。其实 EndHi() 方法也是先复制继承了，只是立刻被覆盖了（所以必须用 :function! 命令，加叹号）。在使用时，也就可以直接用子类调用 new() 方法创建 子类对象，再子类对象调用 Hello() 方法。

至于面向对象的多态特征，对于弱类型脚本语言而言，只要实现了继承的差异化与特例化，是天然支持多态的。例如上面的 obj5 与 obj6 虽属性于不同类型，但可直接放在一个集合（如列表）中，然后调用统一的方法：

```
: let lsObject = [obj5, obj6]
: for obj in lsObject
:     call obj.Hello()
: endfor
```

但是对于其他强类型语言（如 C++），却不能直接将不同类型的 obj5 与 obj6 放在 同一个数组中，才需要其他语法细节来支持多态的用法。

小结

VimL 提供的字典数据结构，允许程序员写出面向对象风格的程序代码。在字典内，可同时保存数据与函数（引用），并且在这种函数中可以用关键字 self 访问字典本身，因而可以将字典视为一个对象。类、继承子类与实例化对象，都可以简单通过复制字典来达成。只是这种全复制的方式，效率未必高，因为会将类中的方法即函数引用都复制到子类与实例中，即使函数引用所占空间很小，也会造成一定的浪费。然而，从另一方面想，光脚的不怕穿鞋的，VimL 本来就不讲究效率，这也不必太纠结。几乎所有的语言，面向对象设计在给程序员带来友好的同时，都会有一定的实现效率的代价交换。

第七章 VimL 面向对象编程

7.3 自定义类的组织管理

在上一节已经讲叙了如何利用 VimL 语法实现面向对象编程的基本原理，本节进一步讨论在实践中如何更好地使用 VimL 面向对象编程。关键是如何定义类与使用类，如何管理与组织类代码使之更整洁。因为从某种意义讲，面向对象并不是什么新的编程技术，而是抽象思考问题的哲学，以及代码管理的方法论。

笔者在 github 托管了一个有关 VimL 面向对象编程的项目 vimloo，可作为一个实现范例。本节就介绍这个 vimloo 项目的基本思路，不过该项目代码有可能继续更新维护与优化，故本节教程所采用的示例代码为求简单，不尽与实际项目相同。

每个类独立于一个自动加载文件

在上一节的示例代码中，我们定义了一个名为 class 的类。因为彼时只关注实现原理，并未指定相关代码应保存何处。你可以放在任一个脚本中，甚至也可以粘贴入命令行，也能起到演示之用。

如果你想用 VimL 实现一个规划不太大的（插件）功能，又想用到字典的对象特征，想在单文件中实现全部（或大部分）功能，那么也着实可以就像是上节的示例那样，在单文件中定义类然后使用类。但是，既然想到要用面向对象的设计，那么一般地每个类都应该是相对独立完整的功能单元。这时，将类的定义代码提取出来放在独立的文件中就更合适了，这也可以达到隐藏类实现细节的目的，在其他需要使用对象的地方，只需创建相应类的对象，调用该类对象所支持的方法即可。

简言之，要区分类的实现者与使用者（尽管很多时候这是同一个程序员的工作）。在 VimL 中，如果要将类的定义代码单独存于一个文件中，最适合的地方应该就是 autoload/ 子目录下的自动加载文件了。因为它可以让用户从任意地方调用，并且只在真正需要用到时才加载类定义代码。

于是，将上节的 class 类定义稍作修改，保存于某个 &rtp （如 ~/.vim）的 autoload/class.vim 文件中：

```
" File: ~/.vim/autoload/class.vim

let s:class = {}
let s:class.name = 'class'
let s:class.version = 1
```

```
function! s:class.string() abort dict
    return self.name
endfunction

function! s:class.number() abort dict
    return self.version
endfunction

function! s:class.disp() abort dict
    echo self.string() . ':' . self.number()
endfunction
```

主要是将定义的类（字典）名字改为 s:class，使之成为局部于脚本的变量。这样在不同文件中定义的不同类也都能用相同的字面名字 s:class 而互不冲突。该变量名的选用是任意的，在不同类文件中选用不同变量名也可以，只要在随后定义类的属性与方法也都用相应的字典变量名即可。但这里的建议是，为求风格统一，每个类文件定义的类字典变量都取名为 s:class。

在这个 class.vim 中定义的类没打算做什么实际工作，因此只（貌似随意地）定义了两个属性与几个方法。当然，你也可以将 string() 与 number() 方法想象为类型转换方法，用于在必要时如何将一个对象转为字符串或数字的表示法。

使用自动加载函数处理类方法

现在 class.vim 文件中定义的 s:class 类只能在该文件中访问，这显然是不够的。为了达到分离类定义与类使用的设计原意，我们还得在 class.vim 提供一些公有接口让外界使用类。自动加载函数就是一个很好的选择，因为它既是全局函数，又通过 # 前缀限定了“伪作用域”。例如，添加以下函数：

```
" File: ~/.vim/autoload/class.vim

function! class#class() abort
    return s:class
endfunction

function! class#new() abort
   let l:obj = copy(s:class)
   return l:obj
endfunction

function! class#isobject(that) abort
    return a:that.name ==# s:class.name
endfunction
```

先看 class#class() 这个略有奇怪的函数命名。class# 前缀部分是对应 class.vim 文件名路径的，class() 可认为是该函数的基础名字。它的作用很简单（也很关键），就是返回当前文件定义的类 s:class，使外界有个途径能使用这个类。这

就是个取值函数，也可命名为 getclass() 或许可更易理解。

class#new() 函数就是用于创建一个新对象。我们使用一个类时，第一步往往就是新建对象，这就只要调用 class#new() 就可以了。如果之前尚未加载类 class 的定义， 就会按自动加载机制加载 class.vim，也就完成其内 s:class 的定义。普通用户一 般情况下根本用不到 class#class() 获取 s:class 的定义，除非想动态修改类定义（慎重）。如果真的想向用户完全隐藏类定义，不提供 class#class() 函数即可，只 提供 class#new() 让用户能创建对象好了。

所以才将创建对象的函数定义为 class#new() 而非像上节那样的方法 s:class.new()，让用户直接上手创建对象，而不必关心类定义是否已加载。其次也是由于 VimL 只能按复 制式创建对象，如果把 s:class.new() 方法也复制到对象中，是很没必要的，甚至还可能被误用。

至于 class#isobject()，用于判断一个对象是否属于本文件所定义的类。在某些应用中，先作类型判断是有意义的甚至是必要的。这里暂且先用类的 name 属性来标记一个类，因此为了保证类名的唯一性，name 属性的取值也按自动加载函数的规则取文件名路径（即如 class#class() 函数的前缀部分）。如果在某个深层子目录中定义的类，如 autoload/topic/subject.vim 文件内定义的 s:class 类名属性就应该是 topic#subject。当然了，另有一个建议，由于 VimL 的大多数脚本都未必是类定义文件，为了更明确表示它是个类文件，可将更多实用的类都统一放在 class/ 子目录下， 如 autoload/class/topic/subject.vim，如果其类名就是 class#topic#subject。 严格地讲，class#isobject() 要稳健地执行，还应判断所传入的参数 a:that 是否 字典类型，以及是否有 name 这个属性。

然后，可以根据需要设计更多的函数。这有两种选择，如果是操作对象的方法，应存入 s:class 字典，如 s:class.method()。如果它不适合用作对象的方法属性，而着重与类型有关，可定义为自动加载函数，如 class#func()。

区分类属性与对象属性

从前面的章节讨论中，我们意识到类属性与对象属性可以是两个不同的概念，这是值得优化的一个方向。尤其是 VimL 中若用简单粗暴的全复制方式创建对象，把那些通用的属性复制到每个对象中，显然是个浪费。例如上一小节的类名属性 name，尤其是深层目录的类文件，像 class#topic#subject 这样的字符串已经不短了，每创建一个新对象都保存这样一个属性值，似乎很不值了。

但另一方面，在类定义字典中保存类名属性也是有意义的，因其关联了文件路径，也可据此间接调用方式文件内的自动加载函数。所以，最好是能限定类名属性不被复制到新建对象中。因此为了区分，约定将类属性的命名加两个下划线，如 _name_。这样，某些具体的对象也可能需要自己的 name 属性，也不致键名冲突。

按这种思路，我们再试写另一个类文件：

```
" File: ~/.vim/autoload/class/subclass.vim

let s:class = {}
let s:class._name_ = 'class#subclass'
let s:class._version_ = 1
```

```
" Todo:
let s:class.value = 0

function! class#subclass#class() abort
    return s:class
endfunction

function! class#subclass#new() abort
   let l:obj = Copy(s:class) " Todo:     " "
   return l:obj
endfunction
```

之前定义在 autoload/class.vim 文件中名为 class 的类，不妨当作整个自定义 VimL 类系统的通用基类。在实际工作中一般不会直接用到 class 类及其实例对象。所以我们开始设计实际可用的子类，建议将所有实用类归于 class/ 子目录下。以上也仅是个说明示例，故类名简单取为 subclass，按自动加载机制，其全名则是 class#subclass 。

这个类文件的基本框架与之前类似，只不过将原来的类属性改名为 _name_ 与 _version_ 。属于该类的对象的属性名，不加下划线，比如 value。然后创建对象的 #new() 函数，显然不能直接用 copy() 或 deepcopy() 内置函数了。这个辅助的特殊复制函数需要另外实现，不过将其命名为 Copy() 或 SpecialCopy() 就显得有 点蠢了。联想到之前的 class#new() 函数，既然一般没必要创建 class 顶层基类的 实例对象，不妨将 class.vim 内定义的函数改为公共基础设施函数。于是修改如下：

```
" File: ~/.vim/autoload/class/subclass.vim

function! class#subclass#new(...) abort
   let l:obj = class#new(s:class, a:000)
   return l:obj
endfunction
```

这里，只是将当前文件定义的类 s:class 与任意参数 a:000 传给 class#new() 基础设施函数，然后也是返回所创建的对象。至于 class#new() 的具体实现，略复杂，请参考 vimloo 项目的 autoload/class.vim。这里只说明它主要做的几件事：

一是分析 s:class 的键，过滤掉带下划线前后缀的属性名，只把普通属性复制到对象实例中。如上例的 class#subclass 类，由 #new() 创建出的对象只有 value 属性。

二是给每个新建对象添加唯一一个特殊属性，名为 _class_ ，就是对 s:class 的引 用。这样每个对象都能知道自己所属的类了，在有必要时可访问这个类字典获得其他信息。而且保存类字典的引用，比保存类名字符串在安全性与效率性上都好得多。然后，判断一个对象是否属性本类的函数也能利用该属性，可大约修改如下：

```
" File: ~/.vim/autoload/class/subclass.vim

function! class#subclass#isobject(that) abort
```

```
    " is       ==
    return type(a:that) = type({}) && get(a:that, '_class_', {}) is s:class
endfunction
```

其实还有第三个隐藏事件，这只在每个类创建第一个对象时发生。为了避免每次创建对象都要作第一步的分析过滤 s:class 的键名，class#new() 会在第一次记忆这个结果，保存在一个特殊键 s:class._object_ 中。这是向用户隐藏的第一个实例，用户新建使用到的实例是直接从这个实例深拷贝的（deepcopy()）。我们可以将其视为这个类的“长子”，是其他实际干活的小弟们的楷模。

控制继承与多层继承

然后讨论 vimloo 项目对继承的实现。首先不要惊讶于命名学上的选用。因为前文已经说明，继承与实例化一样底层都是通过复制实现的。既然创建新对象是用 #new() 函数， 那么创建新子类就用个相对的单词 #old() 。

假设要从 subclass 继承一个类 subsubclass，类文件保存于 class/subsubclass.vim。当然你也可保存于 class/subclasss/subsubclass.vim 文件中，只是名字略长。这里要指出的是，文件系统的目录层次，未必要强求与类的继承链一一对应，那也会有其他麻烦，仅从文件管理角度看，将相关主题的类文件放在一个目录中就能接受了。

要实现这个继承关系，有两点需要改动。一是在 subsubclass.vim 中创建 s:class 时不再初始化为空字典，而是调用 subclass#old() 返回的字典；二就是要在 subclass.vim 中实现 subclass#old() 函数，描述如何将自己这个类继承（复制）给子类。代码框架如下：

```
" File: ~/.vim/autoload/class/subsubclass.vim

let s:class = subclass#old()
let s:class._name_ = 'class#subsubclass'
let s:class._version_ = 1

" Todo:
function! class#subsubclass#new(...) abort
   let l:obj = class#new(s:class, a:000)
   return l:obj
endfunction
```

```
" File: ~/.vim/autoload/class/subclass.vim

"      #old()
function! class#subclass#old(...) abort
   let l:class = class#old(s:class)
   return l:class
endfunction
```

可见 subsubclass.vim 的类定义框架与之前的 subclass.vim 很是类似，只有第一行初始化 s:class 的不同。甚至创建对象的 #new() 方法的写法也完全一样，因为把复制的细节都提炼到 class#new() 这个通用设施上了。用户可直接上手调用

class#subsubclass#new() 方法创建对象，按 VimL 自动加载机制，subsubclass.vim、subclass.vim 与 class.vim 这三个脚本文件都会触发加载。

至于继承函数 class#subclass#old() 与实例化函数 class#subclass#old() 也类似 ，将复制的细节委托通用的 class#old() 函数处理。它也是分析过滤 s:class 的键 ，将必要的键复制给子类，并在子类字典中添加一个特殊键 _mother_ 引用自身类字典 。（具体实现代码就不帖了，看 vimloo 项目源码）

如果要让 subclass 继承自 class，也可修改 subclass.vim 中对 s:class 的创建语句 let s:class = class#old()。因为 class#new() 与 class#old() 函数 接收可变参数，一般将其第一个参数视为类定义字典，即其他类文件中的 s:class，当 然也可以是类名字符串，根据类名可获取其 s:class 字典；如果没有参数时，就用 class.vim 文件本身的 s:class 类字典。不过，由于在 class.vim 的 s:class 在实践中实在乏善可陈，在第二版（_version_ = 2）时，无参调用 class#old() 会 快速返回空字典 {} 。自定义的顶层类没有母类（基类），或 _mother_ 属性为空。

因此，vimloo 实现的类体系，可类比“母系社会”来理解。从一个母类中有两种繁衍，“女儿”是子类，主要用途就是继续繁衍；“儿子”是实例化对象，就是用来实际工作干活的。子类中通过 _mother_ 属性记录母类的联系，实例中是 _class_ 属性。由于实际工作中可能需要许多同质的实例对象，故而还设置了一个隐藏的 _object_ 长子监管。这套机制用于描绘单继承应该足够清晰易懂。

能用单继承解决的问题，尽量避免多重继承。不过 vimloo 也实现了多重继承的支持。每个类的 _mother_ 属性虽然只记录了唯一的母类，但也允许有其他基类，有两种“其他基类”。一种叫 _master_ （意为“师父”），只继承其方法，不继承其数据；另一种叫 _father_ （意为“父亲”），只继承其数据，不继承其方法。每个类的 _master_ 与 _father_ 属性（如果有），都是数组，即可以是多个来自其他类文件定义的 s:class。 只不过这些“其他基类”的属性，都不会直接导入当前文件的 s:class 中，只有当创建 对象实例时（如 s:class._object_ 长子），才会分析这些类的键名，将必要的键复制下来。

也可以通过形象的比喻来理解这个模型。如一位母亲抚育孩子，额外聘请多位老师教孩子其他技艺，这是可理解的（相当于某些语言的接口方法），不过母亲本身未必要掌握这些技能，她的目的是孩子们能学会就可以了。当然了，另一方面，也允许多个“父亲”，这思想有点危险啊，最好避而不用吧。

构造函数与析构函数

重新审视一下创建对象的 #new() 方法，其流程应该要包含以下三步工作：

1. 复制类字典
2. 初始化对象属性
3. 返回对象

其中，第一步与第三步的工作，对于每个类而言，都几乎是一样的，所以在 vimloo 中将 其提炼为 class#new() 函数，可为每自定义类处理通用事务。但是第二步的初始化，显然是每个类有独立需求的。因此，建议每个类文件再写个 #ctor() 函数专司初始化，这就叫做构造函数。

仍以上文的 subclass 为例，将其创建函数与构造函数并列展示如下：

```
" 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
```

理论上，#ctor() 函数内的初始化代码插入到 #new() 函数中也是可以的。不过为了保持 #new() 函数的简单统一，同时为了支持其他间接创建对象的需要，故将构造函数 #ctor() 独立出来。需要注意的是，#ctor() 函数不是由当前类文件的 #new() 函 数直接调用的，而是间接由通用的 class#new() 函数调用。不过可变参数 ... 的意 义在这两个函数之间保持一致，即 #ctor() 内的 a:1 与 #new() 内的 a:1 是 相同意义的参数。在构造函数 #ctor() 中，对象已经被创建出来，第一个参数 a:this 就代表这个刚创建的对象。构造函数一般不由用户直接调用，也不必返回值，只要在创建函数 #new() 中返回对象即可。

一般情况下，在自定义类文件中，建议同时提供创建函数与构造函数，各司其职。但是构造函数不是必须的，尤其是对象属性很少，或能接受每个对象都采用相同的初始值。甚至创建函数也不是必须的，因为也能从通用的 class#new() 函数中创建指定类的对象。例如，以下两个语句是等效的：

```
: let obj = class#subclass#new(100)
: let obj = class#new('class#subclass', [100])
```

显然，使用类文件自己特定版本的 #new() 函数创建对象更简洁，意义更明确。不过通用的 class#new() 函数也适用于在程序运行需要动态创建不同类别的对象的情况。如果传入的第一个参数是类名字符串，则相应的类文件中必须定义 #class() 函数（上例 就是 class#subclass#class()） 才能获取其类定义 s:class。此外，要让 class#new() 能正确调用构造函数，也依赖于类字典 s:class 保存了类名属性 _name_。

对于子类的构造函数，写起来略为复杂些。因为你肯定期望能复用基类（母类）的构造函数初始化继承自母类的那部分数据属性。class.vim 提供了一个 class#Suctor() 函 数用于获取一个类的母类的构造函数（引用）。于是 subsubclass 的构造函数可写成如下形式：

```
" File: ~/.vim/autoload/class/subsubclass.vim
function! class#subsubclass#ctor(this, ...) abort
    let l:Suctor = class#Suctor(s:class)
    call call(l:Suctor, extend([a:this], a:000))
    " Todo:
```

```
endfunction
```

其中，call() 内置函数的用法不算简单，请参考文档 :h call()。如果你确知母类的构造函数没有做什么实质性的初始化工作（甚至未提供构造函数），也可以省去调用母类构造函数的步骤。如果硬编码调用母类的构造函数，如 class#subclass#ctor()，也不是不可以，但显然太过僵硬了，且写法上也未必比利用 class#Suctor() 省多少。在上例中，直接将所有的参数 a:000 传给母类的构造函数处理。在实践中，可能只需要部分参数传给母类，如果这部分参数正好是可变参数的前面几个，那么直接传 a:000 也可能是正常的。在其他其他情况下，可能要对参数作某些预处理再传给母类的构造函数。

在那些没有自动回收垃圾机制的面向对象语言（如 C++）中，与构造函数相应地，还有析构函数。VimL 脚本语言显然是能自动回收垃圾的，不须由程序员作此负担。不过 VimL 在处理有环引用（如双向链表、树、图等复杂结构）中，垃圾回收会有滞后。为此，也可以在自定义类文件中写个“析构函数”，命名为 #dector()，用于打断对象内部的环引用。当确实用不到一个对象时（往往是在函数末尾），调用 class#delete(object)，它会自动调用相应类文件的 #dector() 方法，然后当这个对象离开作用域时，就能立即被回收了。vim 也有个内置的函数 garbagecollect() 可触发立即回收垃圾，但它可能要用到搜索判断环引用的复杂算法。如程序员能帮它的回收机制打断环引用，也应是善事，尽管这是可选的，不是必须的。

类的外包与简化使用

有了以上讨论的 vimloo 提供的面向对象功能，我们就能根据具体的功能需要，设计自定义的类（体系）了，然后创建对象完成实际的工作。

不过还有个小问题，就是类名可能太长，书写不便。假如有这么个类，全名是 class#long#path#topic#subject。用户在使用这个类时，每次创建对象都得调用 class#long#path#topic#subject#new() 函数。这已经算麻烦的了，如果以后想重构，想对类重命名或移动存放目录路径，那每个创建对象的地方都还得作相应修改，那就不仅麻烦，也更易遗漏出错了。

为此，vimloo 再提供一个 class#use() 函数。先直接看用法示例：

```
" File: ~/.vim/autoload/class/long/path/topic/subject.vim

"

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()
"
" 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
```

简言之，class#use() 创建会创建一个字典，默认情况下有以下几个键：

- class：就是引用在类文件中定义的类字典 s:class
- new ：函数引用，相关类文件的创建函数 #new()
- isobject ：函数引用，相关类文件的创建函数 #isobject()

就是将某个类定义及两个最重要的自动加载函数（的引用）打包在另一个字典中，可以提供额外参数（函数名列表，不含 # 路径前缀）指定打包其他的自动加载函数，但 class 是不需要指定，必然被打包在其内的。由于这仅是作了一层简单的包装，提供给外部使用，故简称为“外包”机制。注意类的方法（如 s:class.method()）是不需要外包的，因为那是通过之后创建的实例对象访问的。

通过这种外包，用户代码就可大为简化了。例如可以在脚本开始将要用到的类的外包保存在一个脚本局部变量，如 s:CPack，然后在该脚本内就可以用 s:CPack.new() 创建该 类的对象了。这是自动加载函数的引用，同样可以触发相关类文件的自动加载。如果此后类名发生了修改，或者就是想试用另一个类，也只要修改开始的一处代码而已。甚至在创建子类时，也可以利用外包书写，如：

```
let s:CPack = class#long#path#topic#subject#use()
let s:class = class#old(s:CPack.class)
"
" let s:class = class#long#path#topic#subject#old()
```

另外要提示的是，class#use() 函数会记录已经被外包使用的类。所以在正常运行时，每个类只会创建一个外包，在多个脚本中使用同一个类的外包时，并不会增加额外的开销。

类文件框架自动生成

从以上内容可感知，创建一个自定义类文件，有着大致相似的框架，主要包含以下几部分内容：

- 创建 s:class 字典，可以是简单的空字典或继承其他类；
- 为 s:class 增加数据属性键，可用初始值约定数据类型；
- 为 s:class 创建字典函数，用作类的方法；
- 提供一些必要的自动加载函数。

为了节省键盘录入字符的工作，vimloo 也提供了一些命令，用于根据模板文件生成类定义文件的基本框架。这可节省 VimL 类开发者的大量工作，通过命令生成基础代码（甚至可以再自定义映射，一键生成）后，只要再填充必要的类定义实现即可。

- `:ClassNew {name}` 当前目录在某个 `autoload/` 或其子目录时可用，提供一个文件名参数，将新建一个 `.vim` 文件，并根据该文件名创建一个类。
- `:ClassAdd` 当正在编辑 `autoload/` 或其子目录下的某个 `.vim` 文件时，用该命令向当前文件添加一个类定义。
- `:ClassPart {option}` 与 `:ClassAdd` 类似，但只根据选项生成部分代码，而非全部代码，用于补遗。

类定义的框架模板文件位于 vimloo 项目的 `autoload/tempclass.vim`，这也是一个符合 VimL 语法的脚本，同时也是个五脏俱全的类定义文件。该文件的每个段落开始有行注释，注释行末尾是类似 `-x` 的选项字符串，其中若小写字母表示默认生成这段代码，大写字母表示不生成这段代码。但以上命令可附加额外选项覆盖默认行为，多个选项字母拼在一起当作一个参数传入。

若使用时还遇到疑问，请参考 vimloo 项目的说明文档或帮助文档。

第八章 VimL 异步编程特性

8.1 异步工作简介

异步机制是 vim8 版本引入的新机制，准确地说，是从 7.4 某个补丁开始引入，不过在 vim8 完善并正式发布。这一全新特性使得 vim 直接跳升一大版本号，可见意义非凡。

8.1.1 同步工作可能的问题

要理解异步的特性，不妨先回顾下在此之前只能同步工作的情况，会遭遇哪些不便。

比如要从一个目录下的文本文件中查找某个字符串，我们知道（在 unix 系统中）直接有个 `grep` 工具可用。而在运行着的 vim 中，也可以通过 `:!grep ...` 命令调用系 统的 `grep` 工具。但是用 `:!` 执行外部命令的话，会临时切回启动当前 vim 的终端，外部命令的输出在该终端上；当外部命令经过或长或短的时间完成后，还需要等用户按回车确认才回到 vim 正常的用户界面。如果是 windows 系统的 gVim，`:!` 执行外部 命令则会弹出 `cmd` 黑框，展示外部命令的输出，也需要由用户确认关闭该黑窗才能回到 gVim 编辑窗口。

显然按种方式，在运行外部命令的同时，在回到 vim 界面之前，vim 对用户而言是停止工作的，比如用户暂时无法操纵 vim 进行编辑工作。vim 也有个类似的内部命令 `:vimgrep` 用于在多文件中搜索字符串，并将结果输出在 `quickfix` 窗口。运行该命令不会切回 shell 终端，与 `:!grep` 很有些不同。但是，如果待搜索的文件很多，尤其是类似 `**/*` 的递归所有子目录的文件搜索时，`:vimgrep` 命令完成搜索也可能很慢，需要等待一段时间才能完成搜索。在等待的这段时间内，虽然仍然停留在 vim 界面，但 vim 也好像停止了与用户的交互工作，譬如按 `j k` 不见得会移动光标。事实上 vim 还是监测到你按了 `j k` 键，只不过要等 `:vimgrep` 这个慢命令完成后才会响应后续按键。简单地说，就可能造成明显卡顿。

这就是旧版本 vim 按同步工作方式可能出现的问题。你可以将 vim 编辑器想象为一个单线程的无限循环程序，等待着用户的按键，并立即根据按键命令处理工作。正常情况下 vim 响应用户按键命令是极快的，所以用户感觉很流畅。因为正常人类的击键速度在计算

机程序看来都太慢了，vim 在大部分时间里都是在等待用户击键的。但是当用户试图让 vim 执行某些“能感觉出来慢”的命令时，问题就浮现了，影响用户体验。

如果上面的 :vimgrep 命令没让你感觉到慢，可以用 VimL 定义如下的慢函数：

```
function! SlowWork()
    sleep 5
    echo "done!!"
endfunction
```

然后在命令行输入 :call SlowWork() 并回车，你应该就能感觉到 vim 明显卡顿了。 在此期间若按几次 j ，也要等该函数返回才能发现光标移动。此外，你也可以试试用 while 1 | endwhile 定义一个无限循环函数，调用时会令 vim 完全停止响应，如此请用 Ctrl-C 强制结束当前命令，回到 vim 的正常工作状态来。

8.1.2 异步工作想解决的问题

显然，vim8 引入的异步机制，就是试图解决（或部分解决、缓和）上述同步模型中出现的“慢命令卡顿”问题。当然它也不是直接重定义优化原来命令的工作方式，因为兼容旧习惯也是 vim 的传统。所以，在 vim8 中，类似 :!grep 或 :vimgrep 命令，该怎么慢还怎么慢，它真正想优化的类似 system('grep') 函数的工作方式。

system('grep') 与 :!grep 的相同之处在于都是调用外部命令（系统可执行程序），只不过调用 system() 函数不会切到 shell 终端，仍停留在 vim 界面。所调用的外部命令的输出会被 system() 函数所捕获，可以保存在 VimL 变量中，供脚本后续使用。如果该外部命令执行时间较长， vim 用户仍会感到停止响应或卡顿。

然后在 vim8 中，就提供了另一套不叫 system() 名字的函数，用于执行外部命令。vim 不再等待外部命令结束，而是立即返回给用户，可以立即接着响应用户按键。等外部命令终于结束了，vim 再调用一个回调函数处理结果。

开启异步工作的具体函数与用法，留待下一切详细介绍。不过你该能感觉与估计到，这个异步编程模型比本书之前介绍的同步编程模型要复杂些。并且在监测外部命令结束时准备回调也必然有其他开销，所以异步也不宜滥用，只适合在（可预期）比较耗时的外部命令上。如果只是简单的可以快速完成的外部命令，仍用原来的 system() 函数完成工作即可。

另外要提及的是，目前 vim8 版本的异步机制，也只能将外部命令以异步的方式开启，并不能用异步的方式执行内部命令。也就是说，不论是 vim 内置的命令（及常规函数），还是用 VimL 写的自定义命令（函数），都仍只能按原来的同步方式执行，暂无异步用法。

8.1.3 异步机制带来的 vim 新特性简介

vim8 提供异步机制后，可以据此实现很多新特性。比如内置终端（从 vim8.1 版本开始 支持）。在命令行执行 :terminal 就能打开一个新窗口，体验一下内置终端。在这个特殊的 vim 新窗口中，就相当于运行着一个 shell ，可以像系统 shell 一样执行任何 命令，甚至也可以在此又运行一个 vim （不过一般情况下不建议这么玩）。用窗口切换快捷键 Ctrl-w 可以回到之前的普通 vim 窗口，正常操作 vim 进行编辑工作。

也就是说，内嵌终端正是异步运行的，并不中断 vim 本来的编辑工作。相比在这个功能出现之前，用 :shell 命令打开的子终端，就会切出 vim 界面，只能在那个子终端中工作，必须在那执行 $ exit 退出子终端，才能回到 vim 。

关于内置终端的详细用法请参考 :help terminal ，在那文档中还介绍了在 vim 中“ 嵌入”gdb 调试 vim 本身的示例。表明内置终端功能其实不止能执行一个 shell ，还 适于执行其他任何交互程序，例如 python 解释器，mysql 客户端，gdb 调试器等。

不过本章不是介绍 vim 的这类新特性，而是侧重介绍 VimL 脚本编程中如何使用这个异步机制，据此可以完成之前的脚本无法完成的工作，或优化某些插件功能。

8.1.4 异步编程的简单运用：定时器

让我们先看一个简单的例子来体验下异步编程的风格，定时器（请确认 vim 编译包含 +timers 特性）。将上文按传统同步风格定义的 SlowWork() 函数重新改写如下：

```
function! SlowWork()
    call timer_start(5*1000, 'DoneWork')
endfunction

function! DoneWork(timer)
    echo "done!!"
endfunction

call SlowWork()
```

现在再调用 SlowWork() 函数时就不会“暂停” 5 秒了，该调用立即返回，用户可如常操作 vim 。大约过了 5 秒后，函数 DoneWork() 被调用，显示 "done!!" 。

这里的关键是在 SlowWork() 中用 timer_start() 启用了一个定时器。参数一是时间，单位毫秒；参数二叫回调函数，应该是函数引用，但也可用函数名代替。其意义就是在指定时间后调用那个回调函数，而不影响现在 vim 对用户的正常响应。还可以指定可选的第三参数，表示重复回调若干次，默认就只回调一次，然后自动关闭定时器。该函数有返回值，表示定时器 ID，在 vim 内部就用该 ID 标记这个定时器。回调函数一般是自定义函数，必须接收一个表示定时器 ID 的参数。不过在这个简单示例中，我们忽略未用到这个定时器 ID 参数。定时器相关函数的详细用法请参考 :help timer-functions 。

由此可见，异步编程的基本思路是将原来在一个函数内的工作（一般是较费时的工作），多拆出一个回调函数，用来在工作完成时处理“后事”，关键也就是回调函数的编写。在这个例子中，我们用定时器来“模拟”了一件慢工作，当然定时器本身也另有用途场景。

定时器可以明确指定延时几秒，不过在实际的慢工作（外部命令）中，需要多长时间完成工作是不确定的。这就需要另外的机制，根据其他条件来调用回调函数。这就是下一节准备讲的“任务”，原文档术语叫 job 的话题了。

第八章 VimL 异步编程特性

8.2 使用异步任务

注意：本节所介绍的功能要求 vim 编译包括 +job 特性。

8.2.1 简单任务体验

前文说到，Vim 的异步任务主要是针对外部命令的。那我们就先以最简单最常见的系统命令 ls 为例，其功能是列出当前目录下的文件，若在 Windows 操作系统下或可用 dir 命令代替。

首先请在 shell 中进入一个非空目录，便于实践，并在 shell 中执行如下命令：

```
$ ls
```

然后启动 vim 中，在 vim 命令行中执行如下命令：

```
:!ls
```

体验一下 vim 直接执行外部命令的现象。与在 shell 中执行几乎是一样的，只是将输出打印到终端，供用户交互时查看。然而在用脚本编程中，我们一般希望将外部命令的输出保存到某个变量，便于后续控制与利用。如此可用 system() 函数：

```
: let g:dir_list = system('ls')
: echo g:dir_list
```

当然一般而言，ls 命令执行得足够快，在 VimL 脚本中能很快捕获到其输出。不过我们暂时忽略外部命令的速率，再来看来如何用异用任务完成类似的任务。

```
function! OnWorking(job, msg)
    echomsg 'well work doing:' . a:msg
    let g:dir_list .= a:msg . "\n"
endfunction

function! DoneWork(job)
    echomsg 'well work done:'
    echomsg g:dir_list
endfunction

function! StartWork()
    let g:dir_list = ''
    let l:option = {'callback': 'OnWorking', 'close_cb': 'DoneWork'}
    let g:job_ls = job_start('ls', l:option)
endfunction
```

在这个示例中，函数中直接使用全局 g: 变量，并非良好编程规范，这里仅作说明目的，便于在命令中测试观察。 在命令中输入 :call StarWork() 运行示例。

内置函数 job_start() 用于开启一个异步命令。其第一参数就如同 system() 函数 的参数，指定要运行的外部系统命令。第二个可选参数是个有诸多键的字典，用于配置或

控制该任务的形为。其中最重要的参数就是设置回调参数，在该示例中指定了两个回调函数。一个是 OnWrking() 在工作进行时调用，每当所执行的任务有输出时就会被调用，输出会通过第二参数传入回调函数；另一个是 DoneWork() ，在工作完成时调用。当然应该知道，这两个函数名是我们任意自定义的，名字不重要，关键的魔法是键名，callback 与 close_cb 标识了对应的函数（引用）在适当的机会被调用。

这两个回调函数为求简单，忽略了第一个作为任务标识的参数，并且仍利用全局变量 g:dir_list ，在工作进行时将外部输出收集（串接）起来，最后在工作完成时一次性地将其完整地用 VimL 打印出来，或为其他更有价值的利用。这里用 echomsg 而不是 echo 命令是为了能在随后（通过 :message ）查看消息历史记录。不过要注意，虽然 g:dir_list 在串接时添加了回车符变成多行文本，但 echomsg 仍将其当作一行输出，于是回车符会被其他可打印符号（^@）代替。可手动执行 :echo g:dir_list 再确认它是多行文本，或用 split() 函数将其分隔到列表中。

另一点要注意的是，并非每次 job_start() 启动任务都得注册这两个回调，根据实际工作任务情况可在其中一个（或更多）回调函数中处理感兴趣的信息。甚至如果只是想让某个外部命令在后台默默运行，不关心任何反馈的话，也可以不注册任何回调函数。譬如在后台用 ctags 更新索引文件。只不过提供回调的话，会使异步任务更有交互感与确认感，让用户知道后台命令确实在执行了。

8.2.2 job 选项及其他相关函数

job_start() 的第二参数支持相当多的选项，详情请见 :help job-options ，这里 择其要解释相关概念。帮助主题中所列的选项，不仅给这个 job_start() 函数使用， 也供更通用的底层“通道” ch_open() 利用，后者在下一节继续介绍。现在只需理解，任务（job）是通道（channel）的一种特例或具体应用。

任务采用管道（pipe）将外部命令与 vim 联接起来，那就涉及标准输入、标准输出与标准错误输出这三套件，在其间的消息传递都采用所谓 NL 模式，可以理解为输入输出都按回车分行的字符串。如果某个输出/输入端是有格式的消息字符串（如 json），则可 通过 in_mode out_mode err_mode 分别设定。不过在大多数情况下，使用默认的 NL 模式就适合，且理解更为自然，当然这实际上取决于所调用的外部命令的需求。

本节开始的示例所谓 callback 回调，其实能同时捕获标准输出与标准错误输出，也就是假设外部命令直接在 shell 中执行会打印到屏幕终端的所有可见信息。如果想更精细地区分两者，那就使用 out_cb 与 err_cb 这两种回调，各居其职。

与 close_cb 类似的回调，还有个 exit_cb 回调。从字面上理解，前者是任务关闭时调用，后者是退出时被调用。exit_cb 回调函数比 close_cb 可多接收一个参数，表示任务的状态。

任务管道使用输入输出还可以重定向到文件，或在 vim 中打开的一个 buffer，使用 in_io out_io err_io 及相关的选项设置。如果捕获输出不是最终目的，就可避免在回调函数中将输出保存至 VimL 变量中，直接设置 out_io 输出至 buffer 中呈现更 为直观。

例如，有这么一个命令 tail -f 可用于监控持续增长的日志文件。如果要从 vim 调用 它，在支持异步特性之前，若用 system() 函数，它永远不会返回，那便无用。然而用

job_start() 启动它，再将 out_io 设置为一个 buffer ，就可以达到目的，直接在 vim 中查看增长中的日志。

当然了，还是使用回调函数的工作流更常见，毕竟编程控制上更灵活。如果最终仍想在某个 vim buffer 中展示输出，quickfix 或 localist 或许也是个更好的选择。譬如异步执行 grep （或其他更佳的搜索工具），将结果放在 quickfix 中也适于跳转。

job_start() 也是有返回值的，返回一个标记，代表这个启动的任务，能传递给其他几个任务相关的函数，以指明操作哪个任务。job_stop() 停止指定任务，如果启动的外部命令是设计为死循环永不终止的，也许在 VimL 中就有必要用该函数显式终止任务了。job_status() 用于查询一个任务的状态：fail 表示任务根本就没成功启动；run 表示 任务正常进行中；dead 表示任务跑完了。job_info() 则可查询有关任务更详细的信息。

一般来说，任务的选项是要在启动时设置，但也有些选项可以在启动之后，还处于 run 状态时，使用 job_setoptions() 补充选项。这运用场景就有些受限了。最后，还有个函数 job_getchannel() 用于获得任务底层的通道。

8.2.3 通用异步插件 asyncrun

job 选项与细节繁多，除了帮助文档，另一个绝好的学习方式是参考优秀插件的实现与运用。这里隆重推荐 asyncrun.vim ， 出自国人网名“韦一笑”大神。如果只是使用，它已经封装得很好了，直接使用 AsyncRun 命令即可。如果是想学习异步编程，则该插件也足够轻量，只有一个单文件，也非常适合参考学习。

比如，浏览大概后直接搜索 job_start 看它是如何启动异步任务的，摘录关键代码如下：

```
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
    let l:options['in_io'] = 'buffer'
    let l:options['in_mode'] = 'nl'
    let l:options['in_buf'] = s:async_info.range_buf
    let l:options['in_top'] = s:async_info.range_top
    let l:options['in_bot'] = s:async_info.range_bot
endif
```

```
let s:async_job = job_start(l:args, l:options)
let l:success = (job_status(s:async_job) != 'fail')? 1 : 0
```

可见，它首先是详细构建选项字典，关键的回调函数显然是引用脚本私有函数的。注意在那个条件分支中设定 in_io 标准输入选项，那是在指定选区时运行 :'<,'>AysncRun 时传入的，把当前 buffer 选定的行供给任务的标准输入。在 job_start() 之后，再 立即调用 job_status() ，可判断任务是否成功启动过。

然后按图索骥，跟踪赋给选项的变量从哪里来，回调函数处理又到哪里去（也正是添加到 quickfix 窗口中）。除此之后，就如常规的 VimL 编程了。

你可以利用该插件体验一下在 vim 中直接执行 make 编译或 grep 搜索：

```
: AsyncRun make
: AsyncRun grep ...
```

对比体验一下在 vim7 之前没有异步支持时只能用类似如下的命令：

```
:! make
:! grep ...
```

8.2.4 小结

异步任务只是 vim8 开始引入的新机制，为解决某些问题尤其是调用外部耗时命令是提供另一种编程模式。要真正利用好异步机制，自然还取决于整体的 VimL 编程技术，比如如何有效地管理变量与函数这种基础水平。不过，如果有在其他语言编写过异步回调的经验，改用 VimL 编写异步任务也是类似的思想，就更容易上手些。

第八章 VimL 异步编程特性

8.3 使用通道控制任务

8.3.1 通道的概念

Vim 手册上的术语 channel 直译为通道，比起任务 job 听来更为抽象。上一节介绍的任务，直观想起来，即使不是瞬时能完成的“慢”命令，也是一项“短命”的命令，可以期望它完成，也就完成了任务。

显然，我们可以用 job_start() 同时开启几个异步命令，但是如果企图通过这方式开启一组貌似相关的任务，可能达不到目的。因为开启的不同任务相互之间是独立的，各自独立在后台运行。比如，连续开启以下两个命令：

```
: call job_start('cd ~/.vim/')
: call job_start('ls')
```

这两条语句写在一起，并不能（或许有人想当然那样）进入目标目录后列出文件。第一条语句开启一个后台命令 cd 进入目录，但是什么也没干就完成了；第二条语句开启另一个独立的后台命令 ls 仍然是列出当前目录的文件。

不过这个需求在 vim 中是有解决办法的，想想在 vim8.1 中随着异步特性增加的内置终端的功能，显然是可以通过开启内置终端，在此内置终端中输入 cd ls 命令列出目

标目录下的所有文件：

```
: terminal
$ cd ~/.vim
$ ls
```

既然可以在 vim 中列出一串内容，可想而知也有他法将列出的内容捕获到 VimL 变量中， 再进行想要的程序逻辑加工。

:terminal 命令其实有个默认参数，就是异步开启一个交互 shell 进程（如 bash）， 只不过这个任务与上一节介绍的异步任务有所不同，特殊在于它是不会主动结束的，相当于一个无限死循环等待用户输入，再解释执行（shell 命令）给出回应。那么 vim 与后 台异步开启的这个 shell 进程（任务），肯定是该有个东西连着，以促成相互之前的通讯，这个东西就叫做“通道”，也就是 channel 。

通道的一端自然是连着 vim ，另一端一般连着的是能长期运行的服务程序。上一节介绍的异步任务，也是有个通着连着外部命令的，如此 vim 才能知道外部命令有输出，什么时间结束，才能在适当时机调用回调函数。只不过那外部命令自然结束后，通道也就断了。所以最好反过来理解，通道才是底层更通用的机制，任务是一种短平快的特殊通道。

Vim 的在线文档 :help channel 专门有个文档来描叙通道（及任务）的使用细节，并且在一开始还有个用 python 写的简单服务程序，用于演示 vim 的通道联连与交互。对 python 有亲切感的读者，可以好好跟一下这个演示示例。从这么简单朴素的服务开始，通道可以实现复杂如内置终端这样的标志性功能。虽然我们学 VimL ，不求一下子就能写那么复杂的高级功能，但理解通道的机制，掌握通道的用法，也就能大大扩展 VimL 编程 的效能，满足在旧版本所无法实现的需求。

8.3.2 开启通道与模式选项

要开启一个通道，使用 ch_open() 函数，我们将其函数“原型”与前面两节介绍的定时器、任务的启动函数放在一起对照来看：

- 定时： timer_start({time}, {callback} [, {options}])
- 任务： job_start({command} [, {options}])
- 通道： ch_open({address} [, {options}])

定时器的第一参数是时间，因为它是将在确定的时间内执行工作，同时定时器要有效用，也必须在第二参数处提供回调函数，以表示到那时执行具体的动作。而任务，是无法提前得知执行外部命令需要多少（毫秒）时间的。所以启动任务的第一参数，就是外部命令，有时这就够了，只要让它在后台默默完成即可；之后的选项是可选的，而且对于复杂任务，也可能需要几种不同时机的回调，故而全部打包在一个较大的选项字典中，令使用接口简单清晰。

至于通道，它更抽象在于，它其实不是针对具体命令的，而是针对某个“地址”，就如 socket 编程范畴的“主机:端口”的地址概念。Vim 的通道就是可以联接到这样的地址，与其另一端的服务进行通讯，至于另一端的服务是由什么命令、由什么语言写的程序，这不需要关心，也不影响。

在通道的选项集中，除了同样重要的回调函数外，还有个更基础的模式选项须得关注，就是叫 mode 的。模式规定了 Vim 与另一端的程序通讯时的消息格式，粗略地讲，可直

观地理解为传输、读写的字符串格式。共支持四种模式，上一节介绍的由 job_start() 启动的任务默认就是使用 NL 模式，意为 newline ，换行符分隔每个消息（字符串）。这里使用 ch_open() 开启的通道默认使用 json 格式。json 是目前互联网上很流行的格式，vim 现在也内置了 json 的解析，所以使用方便灵活。

另外两种模式叫做 js 与 raw 。js 模式是与 json 类似的、以 javascript 风格的格式，文档上说效率比 json 好些。因为 js 编码解码没那么多双引号，以及可省略空值。 raw 是原始格式之意，也就是没任何特殊格式，vim 对此无法作出任何假设与预处理，全要由用户在回调函数中处理。

至于在具体的 VimL 编程实践中，该使用哪种模式的通道，这取决于要连接的另一端的程序如何提供服务了。如果能提供 json 或 js 最好，要不 NL 模式简单，如果边换行符也不一定能保证，那就只能用 raw 了。如果另一端的程序也是由自己开发，那掌握权就更大了，如果简单的可以用 NL 模式，复杂的服务就推荐 json 了。

模式之所以重要，是因为它深刻影响了回调函数的写法。比如 vim 从通道中每次收到消息，就会调用 callback 选项指定的函数（引用），并向它传递两个参数；故回调函数一般是形如这样的：

```
function! Callback_Handler(channel, msg)
    echo 'Received: ' . a:msg
endfunction
```

其中第一参数 a:channel 是通道 ID ，就是 ch_open() 的返回值，代表某个特定的通道（显然可以同时运行多个通道）。第二参数 a:msg 所谓的消息，就与通道模式有关了。如果是 json 或 js 模式，虽然 vim 收到的消息初始也是字符串，但 vim 自动给你解码了，于是 a:msg 就转换为 VimL 数据类型了，比如可能是富有嵌套的字典与列表结构。如果是 NL 模式，则是去除换行符的字符串；当然如果是 raw 模式，那就是最原始的消息了，可能有的换行符也得用户在回调中注意处理。

8.3.3 通道交互

与任务不同的是，通道仅仅由 ch_open() 开启是不够的。那只是建立了连接，告诉你已经准备好可以与另一端的程序服务协同工作了。但一般它不会自动做具体的工作，需要让 vim 与彼端的服务互通消息，告诉对方我想干什么，请求对方帮忙完成，并（异步或同步地）等待回应。虽然有些服务可以主动向 vim 发一些消息，让 vim 自动处理，但毕竟有限，你也不能放任外部程序不加引导控制地影响 vim 是不。所以，有来有往的消息传递，才是通道常规操作，也是其功能强大所在。

互通消息的方式，也与通道模式有关。

向 json 或 js 模式的通道（彼端）发消息，推荐如下三种方式之一：

1. call ch_sendexpr(channel, {expr})
2. call ch_sendexpr(channel, {expr}, {'callback': Handler})
3. let response = ch_evalexpr(channel, {expr})

注意前两种写法，直接用 :call 命令调用函数，忽略函数返回值。它单纯地发送消息，异步等待回应；当之后某个时刻收到响应后，就调用通道的回调函数。但是如第二种用法，在发送消息时提供额外选项，单独指定这条消息的回调函数。

于是就要一种机制来区分哪条消息，vim 在发送消息时实际上发送 [{number},{expr}] ，即在消息之前附加一个编号，组成一个二元列表。该编号是 vim 内部处理的，一般是递增保证唯一，{expr} 才是由程序员指定的 VimL 有效数值（或数据结构），并再由 vim 编码成 json 字符串，或 js 风格的类似字符串。通道彼端接收到这样的消息，将 json 字符串解码，经其内部处理后，再由通道发还给 vim ，并且也是由编号、消息体组成的二元列表 [{number},{response}]。在同一请求——回应中，编号是相同的，vim 据此就能分发到对应的回调函数，传入的第二参数也就是 {response} ，不包含编 号的消息主体。 当然，按第一种写法未指定回调地发送消息，收到响应时就会默认分到在 ch_open() 中指定的回调函数中。

至于第三种写法，一般要用 :let 命令获取 ch_evalexpr() 的返回值。这是同步等待，就如 system() 函数捕获输出一样。同步虽然可能阻塞，但优点是程序逻辑简单，不必管回调函数那么绕。在通道已经建立的情况下，如果另一端的服务程序也运行在本地机器， ch_evalexpr() 可能比 system() 快些。因此，如果预期将要请求执行的操作并不太复杂时，可尽量用这种同步消息组织编程。另外，通道也有个超时选项，不致于让 vim 陷入无限等待的恶劣情况。在超时或出错情况下，ch_evalexpr() 返回空字符 中，否则返回的也是已解码的 VimL 数据，如同 ch_sendexpr() 收到回应时传给回调 函数的消息主体。

对于 NL 或 raw 模式，无法使用上面这两个函数交互，应该使用另外两个对应的函数：

1. call ch_sendraw(channel, {string})
2. call ch_sendraw(channel, {string}, {'callback': 'MyHandler'})
3. let response = ch_evalraw(channel, {string})

其中第二参数必须是字符串，而不能是其他复杂的 VimL 数据结构，并且可能需要手动添加末尾换行符（视通道彼端程序需求而论）。

json 与 js 模式的通道也能用 ch_sendraw() 与 ch_evalraw() ，不过需要事先调用 json_encode() 将要发送的 VimL 数据转换（编码）为 json 字符串再传给这俩函数；然后在收到响应时，又要将响应消息用 json_decode() 解码以获得方便可用 VimL 数据。

因此，所谓通道的四种模式，是指通道的 vim 这端如何处理消息的方式，vim 能在多大 程度上自动处理消息的区别上。至于通道另一端如何处理消息，那就不是 vim 所能管的 事了，是那边的程序设计话题。也许那边的程序也有个网络框架自动将 json 解码转化 为目标语言的内部数据，或者要需要手动调用 json 库的相关函数，再或者是简单粗暴地自己解析 json 字符串……那都与 vim 这边无关了，它们之间只是达到一个协议，需要传输一个两边都能正确解析的字符串（消息字节）就可以了。

此外还得辨别另一个概念，通道的这四种解析模式，与通道的两种通讯模式又不是同一层次的东西。后者指的是 socke 或管道（pipe），是与操作系统进程间通讯的更底层的概念，前者 json 或 NL 却是 VimL 应用层面的模式。上一节介绍的任务，由 job_start() 启动的，使用是管道，重定向了标准输入输出与错误；这一节介绍的通道，由 ch_open() 开启的，使用的是 socket ，绑定到了特定的端口地址。然后，在 vim 中，将任务的管道，也视为一种特殊通道。

8.3.4 通道示例：自制简易的 vim-终端

本节的最后，打算介绍一个网友写的拟终端插件： https://github.com/ZSaberLv0/ZFVimTerminal

这应该是在 vim8.1 暂时未推出内置终端，但先提供了 +job 与 +channel 写的插件 ，目的在于直接在 vim 中模拟终端，执行 shell 命令。虽然没有后来 vim 内置终端那 么功能强大，但也颇有自己的特色。关键是还比较轻量，代码量不多，可用之学习一下如何使用 vim 任务与通道的异步功能。借鉴、阅读源码也正是学习任何语言编程的绝好法门。

首先应该了解，作为发布在 github 上的插件，或多或少都会追求某些通用性，于是在插件中就不可避免涉及许多配置，比如全局变量的判断与设置。就像这个插件，它想同时用于 vim 与 nvim ，两者在异步功能上可能提供了略有不同的内置函数接口，然而还想兼容 vim7 低版本下没异步功能时退回使用 system() 代替。

抛开这些“干扰”信息，直击关键代码，看看如何使用 vim 的异步功能吧。从功能说明入手，它主要是提供了 :ZFTerminal 命令，在源码中寻找该命令定义，获知它所调用的私有函数 s:zfterminal :

```
command! -nargs=* -complete=file ZFTerminal :call s:zfterminal(<q-args>)
function! s:zfterminal(...)
    let arg = get(a:, 1, '')
    " ... ( )
    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
    " ... ( )
endfunction
```

它这里的思路是将开启的任务保存在 b:job 中。这很有必要，因为随后的回调函数都要用到任务 ID （功通道 ID）。它不能保存在函数中的局部变量中，否则离开函数作用域就不可引用该 ID 了，也不宜污染全局变量。于是脚本级的 s: 变量合适；如果异步任务始终与某个 buffer 关联，则保存在 b: 作用域更不清晰，且容易支持多个任务并

行。ZFTerminal 正是将一个普通 buffer 当作 shell 前端来用，因而保存为 b:job 。

如果在执行命令时，任务不存在，就用 job_start() 开始一个任务，否则就向与任务关联的通道用 ch_sendraw() 发送消息。它为这两个函数再作了一个浅层包装（主要为兼容代码考量及定义一些默认选项）。job_start() 它是这样开启的：

```
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',
    \ })
endfunction
```

在这里它指定了几个回调函数，并将通道模式设为 raw 。所以在后续 :ZFTerminal 命令中就用 ch_sendraw() 发送消息了。注意发送消息需要通道 ID 参数，使用 job_getchannel() 函数可以获取相任务关联的通道，并且也保存在 b: 作用域内。 至于回调函数，请自行结合所实现的功能跟踪，此不再赘述。

第八章 VimL 异步编程特性

8.4 使用配置内置终端

8.4.1 使用异步的两个方面

本章讨论的是 vim 的异步特性，其实这包含两个方面。其一如何利用 VimL 编程控制异 步任务，（写插件）实现特定的功能。前三节都是围绕这个话题的，从简单到复杂介绍了 vim 提供的三种异步机制，定时器、任务与通道。那可能有点抽象或晦涩，需要与具体的插件功能结合起来才更好理解，但是基于本书的定位，也不便介绍与解读太复杂的插件。

其二是如何更好地使用 vim 新版本自身提供的异步功能，典型的就是内置终端。作为普通用户，相对于开发，运用可能是更简单有趣的。本节就是打算跳出复杂的异步编程的曲折过程，调剂一下，重新回到简单常规的 VimL 调教与定制内置终端，使之更符合个性习惯，成为日常使用的利器。

当然，这依然是引导性质或经验之谈，详细文档请看 :help terminal 。

8.4.2 内置终端的启动

```
: terminal
: terminal bash
: terminal python
```

用 :terminal 命令开启内置终端。其实广义来讲，它可以接受外部命令参数，在内置终端中运行任意的外部命令，譬如打开一个 python 解释器。默认无参数时就执行

&shell 指定的程序，比如 bash 。

不过一般地，我们提到内置终端，就是指狭义上的在 vim 里面运行一个 shell 。它会横 向分裂一个半屏窗口，在这个特殊的窗口就几乎与外面运行的 shell 一样的操作与功能 ，包括比如 .bashrc 的 shell 配置。

:terminal 除了可以指定外部命令参数外，还可以接受许多选项，控制诸如内置终端的窗口大小、位置等各种选项。你可以将自己的偏好启动选项封装起来，自定义一个函数、命令或快捷键。

此外，除了 vim 命令，还有个 vim 函数 term_star() 用于在编程逻辑中启动一个内置终端，用法就如 job_start() 一样，给予灵活控制，按需启动终端。

8.4.3 终端模式的快捷键映射

在打开的内置终端窗口，为了能像外部 shell 那样使用 shell 本身的快捷键，Vim 禁用 了绝大部分快捷键。虽然在内置终端窗口中可以键入 shell 命令，但那不是 vim 的插入 模式也不是命令行模式，所以 imap 与 cmap 都不生效，当然更不可能是普通模式了 。 事实上，Vim 为此专门新定义了一种特殊模式，叫“终端任务”（Terminal-Job）模式，不妨简称终端模块。如果要为终端模式自定义快捷键，应该用 tmap 系列命令。

不过在动手之前，还是要了解 vim 已经保留了一个特殊键用来切换回 vim 的普通模式； 而且由于前叙原因，也仅保留了一个键。这个键由选项 &termwinkey 给出，默认也是 <C-W> ，因为它的本意正是如何使用 :wincmd 切出终端窗口。于是 <C-W> 引导的 快捷键在终端窗口与普通窗口保持一致的含义，并且附带两个扩展：

- <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> 终端窗口切到普通模式（可以用 hkl 移动了）
- <C-W>: 从终端窗口进入 vim 命令行，（否则按冒号只是在 shell 提示符后输入冒 号呢）

如果不喜欢 <C-W> 这个引导键——比如说因为 <C-W> 在 shell 中是删除前面一个词的快捷键，故想将 <C-W> 键传给 shell ——那么可以设置 &termwinkey 更换。但一 般不建议修改，保持 Vim 内换窗口操作一致性较为重要，况且换任何键都可能会与 shell 冲突，总之是需要权衡。

从终端的任务模式回到普通模式略为麻烦，要按 <C-W><C-N> （在已经按下 <C-W> 的情况下，<C-N> 多按或少按那个 ctrl 键差别不大了）。为什么不保留 <Esc> 键回到普通模式呢？大概是 vim 想兼容更多的终端，有些终端用 <Ecs> 作为转义符。 就个人使用经验而言，在 shell 中会经常用到 <Ecs>. （接一个点）快捷键输入上一 条命令最后一个词。不过权衡之下，可以重定义 <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> 键外，我还定义了其他几个快捷键。比如使用终端时需要经常上翻查看结果，就在 <C-W> 引导键后加个 b ，回到普通模式的同时上翻一页。然后我自己用 tabpage （标签页）比较多，所以也用 <C-W> 加数字切到特定的标签页中。当然，明白了 tnoremap 之后，就能像 nnoremap 一样按自己习惯重定义快捷键了。

另外，按特定方式启动终端也可以自定义便用的快捷键，不过我推荐另一种思路，短命令，例如：

```
command! -nargs=* TT tab terminal <args>
command! -nargs=* TV vertical terminal <args>
```

这意思是用 :TT 命令在另一个标签页打开终端，用 :TV 按纵向分割窗口打开终端。可以将其想象为 <mapleader> 是 : ，而且冒号本来就要按下 shift 键，再接一两个大写字母也顺手，只不过最后还要多按 <CR> 回车确认执行命令。然而这另有个好处是还可以随时增加其他命令行参数（传给 :terminal ），这种灵活性是普通模式下的快捷键不能达成的。因此，“短命令”适合于替代那些“次常用”的快捷键，毕竟键盘布局的快捷键资源以及个人的记忆习惯是有限的。

既然内置终端的启动方式可以定制，那么就想如何能在启动终端时才自动定义那些 tmap 快捷键呢？毕竟 tmap 在平时是用不上，也未必是每次打开 vim 都会用到内置终端，将 tmap （及其他与终端相关的设置）直接写在全局 vimrc 有点“浪费”。vim 显然 也想到了这个需求，很贴心地增加了一个自动命令事件，TerminalOpen 就会在打开内 置终端窗口时触发，于是可将如下事件写在某个合适的事件组（augroup）中：

```
autocmd! TerminalOpen * call OnTermialOpen()
```

将你想要定制内置终端的代码都写在 OnTerminalOpen() 函数中，当然使用 # 形式 的自动加载函数会更好。

8.4.4 内置终端与 vim 交互

所谓交互，自然是分两方面的。其中从内置终端（的任务中）向 vim 发起交互的需求，可能来自一个有趣的“哲学”问题：可不可以在内置终端中输入 $ vim file 再启一个 vim 编辑文件呢。那自然是可以的，但在实用中那显得有点愚蠢，不够优雅。于是，就需要一个机制，从内置终端中向开启它的“宿主” vim 发送消息，令其打开某个文件。

于是 vim 就有了这么个约定（据说来自 emacs），在内置终端的运行的程序，只要向标准输出打印如下序列：

```
<Esc>]51;["drop", "filenmae"]<07>
```

实际上就会将 ["drop", "filenmae"] 传递给宿主 vim ，然后 vim 就知道将该消息解释为执行 :drop filename 命令。:drop 命令其实与 :edit 命令类似，就是打开一个文件，只不过如果文件已被打开，就会跳到相应的目标窗口。:drop 命令也就是随内置终端版本一起增加的，可见它的原意就是想解决这个痛点。

Esc 字符是终端的转义符，在 VimL 中固然可以用 <Esc> 表示，但在其他语言（如 C 语言）中，则一般用 \e 表示，或直接用其 ASCII 码（ \x1B 或 \033 即十进制的 27）表示。

例如，可以在 ~/bin/ 目录下写个简单的 drop.sh 脚本：

```
#! /bin/bash
echo -e "\e]51;[\"drop\", \"$1\"]\x07"
```

注意传给 vim 的消息要求是 json 模式（见前一节的通道模式），drop 与文件名参数 须按 json 标准用双引号括起。在多数语言或脚本中如果用双引号括起整个序列字符串，就得将里面的 json 字符串的双引号用 \" 转义。可以用其他任何语言写这个 drop 脚 本，例如等效的 perl 脚本（dorp.pl）可以如下：

```
#! /usr/bin/env perl
my $filename = shift;
print qq{\x1B]51;["drop", "$filename"]\x07};
```

然后为了使用习惯，可以再在 ~/bin/ 中建个 drop 软链接，指向实用的 drop 脚本 ，如：

```
$ chmod +x drop.pl
$ ln -s drop.pl drop
```

如果 ~/bin 在环境变量 PATH 中，则在 vim 的内置终端中，执行如下命令：

```
vim-shell $ drop file
```

就能在宿主 vim 中用 :drop 打开相应的文件。不过这还有个问题。我们在 shell 中 给任何命令输入文件名参数，一般都是当前目录下的文件名。但是 vim 内置终端的当前 目录，很可能与宿主 vim 的当前目录并不相同，于是 drop 命令可能会失效，所以在传 递消息中应该使用绝对路径，以保证能找到正确的文件。为此，可将原来的 ~/bin/drop.pl 改为如下：

```
#! /usr/bin/env perl
use Cwd 'abs_path';
my $filename = shift or die "usage: dorp filename";
my $filepath = abs_path($filename);
exec "vim $filepath" unless $ENV{VIM};
print qq{\x1B]51;["drop", "$filepath"]\x07};
```

主要改动是利用语言的相关模块获取文件绝对路径，并稍微保护判断下是否是否输入了文 件名参数。另一个改动是倒数第二行 exec ... unless 语句。只有在 vim 的内置终端 才会向宿主 vim 发 drop 消息，如果是从外部普通 shell 使用该脚本，那就会改为启动 vim （进程覆盖当前进程）打开命令行指定的文件，而最后一行再不没机会执行了。从 vim 中启动的内置终端会继承 vim 进程的环境变量，至少它会有 $VIM 这个环境变量 （可以用 :echo $VIM 查看），据此可以判断是内置终端还是外部终端。

当然，如果你熟悉 python ，用 python 写个 drop.py 也是容易的。

在 <Esc>51;[msg]<07> 转义序列中向 vim 传递的消息，除了支持 :dorp 命令，还 支持 :call 命令调用特殊的以 Tapi_ 开头的自定义函数（限定函数名规范是为安全 起见）。消息形如 ["call", "Tapi_funcname", [argument-list]] 。自定义函数约定 接受两个参数，与内置终端窗口关联的 buffer 编号，以及一个参数，所以如果业务逻辑 需要多个参数，就只能将它们打包在一个列表或字典类型的变量，当作一个参数传入。 Vim 开放这么个接口提供灵活扩展的可能，具体能做什么那当然是用户的实现了。

8.4.5 vim 与内置终端交互

交互的另一方面，是 vim 向内置终端发消息。

显示，内置终端也是个任务，有着底层的通道，所以始终可以尝试使用上节介绍的 `ch_sendraw()` 等函数。然而对于内置终端，没必要使用这底层的函数，vim 提供更高 层函数 `term_sendkeys()` 直接向内置终端发送一个字符串，效果如同在终端提示符下手动键入。注意该函数与 `feedkeys()` 的区别，后者是相当于向 vim 键入字符串，会被 vim 截获，并受 `tmap` 映射影响；而前者是直接向内置终端键入，不受 `tmap` 影 响。

试验一下，在打开内置终端的窗口中，使用 `<C-W>:` 进入命令行，输入：

```
: call feedkeys('ls')
```

回车执行后，会在内置终端的提示符之后显示 `ls` 这两个字符，那就是相当于用户通过 vim 界面向内置终端敲了两个字符，但还没敲回车真正发送给内置终端运行。你可以继续编辑这个命令，比如使用退格键删除之，或在其后增加选项 `-l` ，然后再按一次回车，内置终端才能响应执行这个 `ls` 命令。然后，再 `<C-W>:` 试试输入：

```
: call term_sendkeys('', 'ls')
```

发现效果似乎还是一样，`ls` 这两字符停在内置终端提示符之后等待执行。需要将回车键与合在这两个函数的参数中，才是通知内置终端立即执行：

```
: call feedkeys('ls' . "\<CR>")
: call term_sendkeys('', 'ls' . "\<CR>")
```

注意，回车键 `<CR>` 需要双引号转义。并且 `term_sendkeys()` 函数要求第一个参数是指定内置终端的 buffer 编号，空值表示当前内置终端。从用户角度看，如果不涉及（少量的）被 `tmap` 映射的键序列，用这两个函数的效果基本相同，但为了安全起见以及语义明确，向内置终端发消息时，最好用 `term_sendkeys()` 函数。

在上一节介绍的 ZFVimTerminal 插件有个特性，是从 vim 的命令行中向模拟终端发送命令。我们也可以借鉴这个思路，实现从 vim 命令行中向内置终端发送命令。当然了，从内置终端窗口本身再用 `<C-W>:` 进入命令行输命令就有点多此一举了，反而麻烦。所以需求应该是从任何一个普通 buffer 窗口，按 `:` 后在命令行向内置终端发送命令，避免需要跳到内置终端的麻烦；当内置终端不存在时，显然应该打开一个新的内置终端。

为此，可以封装一个函数，并定义命令调用该函数，大致如下：

```
command! -nargs=* -bang TC call useterm#shell#SendShellCmd(<bang>0, <q-args>)
command! -nargs=* TCD call useterm#shell#SendShellCmd(0, 'cd ' . expand('%:p:h'))

function! useterm#shell#SendShellCmd(bang, cmd) abort
    " save current window
    if a:bang
        let l:tab = tabpagenr()
        let l:win = winnr()
    endif
```

```
    let l:found = useterm#shell#GotoTermWin(&shell)
    if empty(l:found)
        :terminal
    endif
    if !empty(a:cmd)
        call term_sendkeys('', a:cmd . "\<CR>")
        " into insert mode to force redraw terminal window
        normal! i
    endif

    " back to origin window
    if a:bang
        if l:tab != 0 && l:tab != tabpagenr()
            execute l:tab . 'tabnext'
        endif
        if l:win != 0 && l:win != winnr()
            execute l:win . 'wincmd w'
        endif
    endif
endfunction "}}}
```

这里，仍然按短命令思想，定义 :TC 用于在内置终端中执行任意命令，就是将其参数用 term_sendkes() 函数数转发给内置终端，并自动添加了回车键。:TC! 加叹号修 饰的话，会回到原来的普通窗口。用 :TCD 跳到内置终端窗口，并自动将内置终端的当前目录切到原来编辑文件所在目录（就是自动执行 cd 命令啦）。因为 TCD 的用意就是切到内置窗口，并开始在指定目录下与终端进行交互工作，那肯定是不必跳回原来的，所以传给实现函数的第一个参数写定为 0 。

查找并切到终端窗口的函数，这里不再列出，主要是通过 &buftype 选项值是否为 terminal 来判断。有兴趣的可以到这个地址查看详细代码： https://github.com/lymslive/autoplug/tree/master/autoload/useterm 。 如果不习惯短命令，或担心命名名冲突，尽可自行改自者再定义个快捷键映射。

第九章 VimL 混合编程

9.1 用外部语言写过滤器

9.1.1 混合编程场景介绍

本章来讨论 VimL 与其他语言混合编程的话题。这“混合”编程可能不是很准确的定义，也许涉及不同层面的场景应用。在上一章介绍的异步编程也算是其中一种吧。不过如果所调用的外部程序是别人已经写好的（或者是系统提供的经典工具），那用户就只能适应其提供的接口或输出，在 vim 端几乎没什么可干预的。但如果利用通道连接的另一端的程序，也要自己开发，那就可以从设计开始就考虑如何更好地与 VimL 协作，并且显然另一端可以使用任何主流语言。这就不再多说了，本章主要着眼于其他场景的（同步）混合编程。

与众所周知的另一件编辑神器相比， vim 是比较纯粹的编辑器，它本身提供的功能（虽

然编辑方面非常丰富）比较集中，也就比较依赖或吻合 Unix 哲学：一个工具把自己的事做好，并且便与其他工具配合。所以，当 vim 想处理更复杂的事务时，它天然地倾向于与其他工具“混用”。比如，从最基本打开文件编辑，vim 也接受从其他工具的管道输入：

```
$ ls -l | vim -
```

这个命令表示将当前目录下的文件列表送入 vim 中编辑，譬如打算在每行前面添加 mv 命令，想仔细规划下如何批量重命名。

就像许多 Unix 工具一样，启用 vim 时若用 - 取代文件名参数，就表示从标准输入读入内容，所以它很容易配合管道，作为接收管道输入的末端。但由于 vim 常规运用是作为可视化交互式全屏编辑，它不再产生标准输出，因而也不便继续产生管道输出至下一工序。然而， vim 也有批量模式，不会打开交互界面，实际上也是可以强行配合，达到类似 sed 的流编辑效果——但这就似乎有点旁门左道了，不是 vim 的常规用法。

当然，管道的配合，只是工具的组合与混用，离“可编程”的概念还比较远。

在支持异步的版本之前，system() 函数只能在 VimL 这端进行逻辑与流程控制，而对所调用的外部命令不可控，这可算“半混合”编程。其实还有另一个叫“过滤器”的功能用法，它是允许与鼓励对所调用的外部脚本进行编程，但在 vim 这端的用法却是固定的，因而也可算是另一种“半混合”编程。

除了自己写通道服务算“全混全”编程外， vim 在这之前还提供了多种脚本语言的内置接口，那也算是（同步的）“全混合”编程了。本节先介绍相对简单的过滤器，下一节再介绍语言接口。

9.1.2 过滤器的概念与使用

即使你对过滤器并不熟，但也应该用过 = 重缩进命令，那就是个特殊的过滤器。

过滤器的意思是将当前编辑 buffer 中指定范围的文本，当作标准输入调用某个外部程序，并当其标准输出替换原范围的文本，以此达到修改、编辑的目的。因此，重缩进与格式化的本质也就是过滤器。

过滤器的标准使用方式是在命令行一对地址范围之后接 ! 与外部命令，如：

```
:n1,n2 !
:1,$ !
:'<, '> !
:. !
```

注意必须在 ! 前有地址参数，否则 ! 就是纯粹切到外部 shell 运行那个外部程序了。而过滤器并不会打断用户切到外部 shell ，只要不是处理巨量文本，替换输入输出应该都比较快，虽然是同步，一般没延迟问题。

如果只有一个地址参数，表示只处理一行， . 表示当前行。如果是两个地址参数，则表示起始行到终止行的范围，1,$ 表示从第一行至最后一行，即全部文本。可以在命令行手动输入两个数字行号，也在可视模式下选择一定范围后按 : 自动添加 '<,'> 表 示所选择的行范围。

使用过滤器还有个快捷键方式，不必先按 : 进入命令行，直接在普通模式按 ! 再接一 个移动命令（文本对象），也会自动帮你选定这个文本对象，并自动进入命令行模式并填

充好地址参数，用户只要继续在 ! 之后输入想调用的外部程序。

诚然，过滤器可以直接调用别人写好、已经完善的外部程序。然而，由于以标准输出替换标准输入的模型如此简单，而每个人的编辑任务又可能多种多样各具个性，在一时找不到合用的外部工具时，用户完全可以用他所熟悉的任一种脚本语言快速写个过滤器。

比如再举那个简单的例子吧，给文本行编号？最简单的需求，其实可以直接用 cat -n 命令完成：

```
:'<,'>!cat -n
```

注意，从 vim 命令行调用外部过滤器时，可以附加命令行参数传给过滤器程序，选择文本是标准输入，这两者互无关系。如果要对全文编号，一定别忘了加地址参数：1,$! ； vim 有不少可能作用于范围的命令，在缺省时默认表示全文，但过滤器若省了地址参数就解释为普通 ! 外部调用命令了。

但是，cat -n 的编号似乎不美观，右对齐，空白太多。如果你想编号左对齐，数字后面最好还能加个符号，如 1. 或 1) 等，再或者想为指定行编号，比如跳过注释行……等等，不一而足的需求。如果你熟悉某种脚本语言，最好是自己操起脚本语言来写适合的过滤器。例如，下面这个 perl 脚本，实现为文本行编号：

```
" file: catn.pl
my $sep = shift || "";
my $num = shift || 0;
$sep .= ($num > 0) ? (" " x $num) : "\t";
while (<>) { print "$.$sep$_"; }
```

该过滤器脚本接受两个参数，第一参数指定紧接数字编号的后缀符号，第二参数指定之后隔几个空白，如果缺省，就隔一个制表符。在 while 循环中，<> 符号用于从标准输入读取数据，$. 表示行号， $_ 表示当前行文本，这样语义就明确了，行号与分隔字符串与原文本拼接起来作为标准输出。（用其他语言写这个脚本也不复杂，只是语法不一样，总是可以手动累加行号的）

如果将该脚本保存在当前目录中，可以在 vim 命令行尝试一下：

```
:'<,'>!perl catn.pl
:'<,'>!perl catn.pl .
:'<,'>!./catn.pl . 2
```

如果给脚本添加了可执行权限，可直接将脚本作为过滤器程序，否则就将脚本文件当作 perl 解释器的第一参数。如果脚本不在当前目录，请替换为脚本全路径，或者若将可执行脚本放在某个 $PATH 路径中，也可以直接使用。然后要注意命令参数，会先后经过 vim 命令行与 shell 命令行两层处理，对特殊字符最好加引号或转义，避免出错。例如：

```
:'<,'>!./catn.pl ')' 2
:'<,'>!./catn.pl '*' 2
:'<,'>!./catn.pl \% 2
:'<,'>!./catn.pl '\#' 2
```

如果) 不加引号，会出现 shell 语法错误；如果 * 不加引号，在 shell 中会展开为当前文件的所有文件名，这可能不是想要的；当然这想两个字符也可以用 \ 转义， 安全地从 shell 命令行传入 catn.pl 过滤器脚本。

Vim 命令行中的 % 符号会被展开为当前文件名，即使用引号，也是将文件名字符串括在引号中传给 shell （如果文件名中有空格，有无引号影响 shell 将其作为几个参数），如果要将百分号传给 shell ，就得用 \% 反杠转义。# 在 vim 命令行中会被展开为“上一个编辑过的”文件名，仅用 \# 可以将 # 传给 shell ，但在 shell 中这符 号是注释，那又会有问题，所以必须用 '\#' 两层保护，才能将 # 符号传入过滤器脚 本中，输出类似 1# 2# 的编号效果。

记不住这许多特殊符号规则怎么办，很简单呀，多试试就好，或者用保守的 '\#' 就差 不多了。而且在 vim 试错了过滤器（参数问题，或脚本本身 bug）不要紧，如果意外修 改了文本，按撤销命令 u 就好。

当然，有时特意利用 vim 特殊符号的替换意义也可能是有用的，例如你又想文件名放在行号之前了，类似 file:1 的效果。那么就可以在 vim 命令行中传入 '%' 参数，如 果确认当前文件名中没空格，也可以不用引号。当然了，这个过滤器脚本本身的逻辑功能也要作相应修改了。

所以你看，只要你经常脑洞大开，需求总是在不断变化。然而只要掌握一门脚本语言，哪怕只会写简单的教科书式的标准输入输出的小程序，运用过滤器思维，就能极大地扩展 vim 的编辑效率与趣味性。

第九章 VimL 混合编程

9.2 外部语言接口编程

9.2.1 语言接口介绍

Vim 支持其他诸多语言接口。这意味着，你不仅可以写 VimL 脚本，也可以使用被支持的语言脚本。这就相当于在 vim 中内嵌了另一种语言的解释器。当然你不能完全像其他语言的解释器来使用 vim ，毕竟还是遵守 vim 制定的一些规范，那就是 vim 为该语言提 供的接口。

在 Vim 帮助首页，专门有一段 Interfaces 的目录，列出了 Vim 所支持的语言接口，大都以 if_lang.txt 命名，其中 lang 后缀指某个具体的（脚本）语言。笔者较熟悉的脚本语言有 lua、python、perl ，而其他如 ruby、tcl 较少了解。因而在本章打算简要介绍下 if_lua if_python 与 if_perl 这几个语言接口。（因 python 有两个版本，故在帮助文档中其实用 if_pyth.txt 命名，避免 python 狭义地指 python2， 不过本文仍习惯使用 python 统称）

一些功能复杂的插件，为了规避 VimL 语言的不足，都倾向于按语言接口采用其他语言来完成一部分或主要功能。比如，unite 就采 用了 if_lua 接口，后来的升级版 denite 则采用 if_python 接口，另外推荐一个插件 LeaderF 也是用 if_python 写的。这都是不错的实际项目源码，想深入学习的可以参考。

不过采用 if_perl 接口的现代插件较少，笔者鲜有看到。但是笔者偏爱 perl ，所以 在本章剩余篇幅将重点以 if_perl 为主，也算略微弥补一点空白。而且，Vim 为各语 言提供的接口大同小异，思路是一致的。介绍一种语言接口，也期望读者能举一反三。真要用好某种语言接口，除了要仔细学习 vim 相关的 if_lang.txt 文档，还需要对目标语言掌握良好，才能方便地在两种环境中来回游弋。

9.2.2 自定义编译 vim 支持语言接口

默认安装的 vim 一般不支持语言接口，需要自己重新从源码编译安装。这也其实很简单，只要修改一些编译配置即可。首先从 vim 官网或其 github 镜像下载源代码包，解压后进入 src/ 子目录， vi Makefile 查找并取消如下几行注释：

```
CONF_OPT_LUA = --enable-luainterp
CONF_OPT_PERL = --enable-perlinterp
CONF_OPT_PYTHON = --enable-pythoninterp
```

原来这几行是被 # 注释的，表示相关语言接口是被禁用的，你所需做的只是删去 # 符号启用功能。当然每个语言接口在 Makefile 都提供了好几个不同的（被注释）选项备用，各有不同的含义，典型的如动态链接或静态链接。上面示例是打开静态链接编译选项，含 =dynamic 的表示动态链接编译选项。你只需打开（取消注释）其中一条选项，一般建议用静态链接编译。动态链接只是减少最后编译出的 vim 程序的大小，或许也略微减少 vim 运行时所需的内存。在硬盘与内存都便宜的情况下，这都不算问题，用静态链接可减少依赖，避免版本不兼容的麻烦。

不过 python 语言接口分 python2 与 python3 两个选项，它们既像一个语言又像两个语言。打开 python3 接口的编译选项是 --enable-python3interp 。注意，你不能同时打开 python2 与 python3 的静态编译选项，如果想同时支持，只能都用动态链接编译选项。除非你有绝对理由想同时使用 python2 与 python3 ，还是建议你只使用其中之一。而且 python2 都是历史原因，以后的趋势都应该都是转向 python3 。

在自定义安装 vim 时，还有个选项推荐打开，就是安装到个人家目录下，不安装到系统默认的路径下，也就不影响系统其他用户使用的 vim 。只要指定 prefix 即可，一般 也就是打开（取消注释）如下这行：

```
prefix = $(HOME)
```

然后，就可以按 Unix/Linux 源码编译安装程序的标准三部曲执行如下命令了：

```
$ make configure
$ make
$ make install
```

如果你运气足够好，应该直接 make 成功的。如果 make 失败，最可能的原因是系统没有安装相应的语言开发包，请用系统包管理工具（yum 或 apt-get）安装语言开发包，如 perl-dev ，注意有些系统为语言开发包名的命名后缀不同，也可能是 perl-devel 。 安装好了所需语言开发包（及可能的其他依赖），再重新 confire make 应该就能成 功了。

在编译成功之后，make install 安装之前，最好检查一下新编译的 vim 是否满足你所 需的特性。执行如下命令：

```
$ ./vim --version
```

在 vim 命令之前添加 ./ 表示使用当前目录（src/ 编译时目录）的 vim 程序，否则可能会查找到系统原来的 vim 程序。如果打印的版本信息，包含 +perl （或 +lua +python），就表示成功编进了相应的语言接口。当然，你也可以直接不带参数地启动 ./vim 体验一下，并可在 vim 的命令行查看如下命令的输出：

```
: version
: echo has('perl')
: echo has('python')
: echo has('python3')
```

:version 命令与 shell 命令参数 --version 的输出基本类似。has() 函数用于检测当前 vim 是否支持某项特性，如果支持返回真值（1），否则假值（0）。has() 函数也经常用于 VimL 脚本尤其是插件开发中，为了兼容性判断，根据是否支持某项特性执行不同的代码。

确认无误后，就可以 make install 安装。所谓安装也不外是将刚才编译好的 vim 程 序及其他运行时文件与手册页等文件，复制到相应的目录中。安装的根目录取决于之前 $prefix 选项，如果按之前指导选择了 $(HOME)，那 vim 就人安装到 ~/bin/vim 中。一般建议将个人家目录下的 ~/bin 添加到环境变量 $PATH 之前，这样在 shell 启动命令时，首先查找 ~/bin 目录下的程序。

当然了，在你决定手动编译 vim 之前，最好在目前默认使用的 vim 中用 :version 与 has() 检测下它是否已经支持相应的特性了，如果已经支持，那就可跳过这里介绍的手动编译流程程了。

9.2.3 语言接口的基本命令

测试某个语言接口是否真的能正常工作，也可直接以相应语言名作为 vim 的命令，执行一条目标语言的简单语句，例如：

```
: perl print $^V
: perl print 'Hello world!'
: lua print('Hello world!')
: python print 'Hello world!'
: python3 print 'Hello world!'
```

其中第一条语句是打印 if_perl 接口使用的 perl 版本，其后就是使用不同语句打印喜闻乐见的 Hello world! 了。

语言名如 :perl 也就是相应语言接口的最基本接口命令了，可见它们保持着高度的一致性，vim 调用相应的语言解释器执行其参数所代表的代码段，所不同的只是各语言的语法文法了。下面，如无特殊情况，为行文精简，就基本只以 if_perl 为例说明了。

基本命令 :perl 只适合在命令行执行简短的一行 perl 语句（当然，对于 perl 语言， 单行语句也可以很强大）。如果要执行一大块 perl 语句，短合在脚本中用 here 文档 语法，即 VimL 也像许多语言一样支持 << EOF 标记：

```
perl << EOF
print $^V; #
print "$_\n" for @INC; #
print "$_ = $ENV{$_}" for sort keys %ENV; #
EOF
```

EOF 只是约定俗成的标记，其实可以是任意字符串标记，甚至可以省略默认就是单个点 . 号。Vim 会从下一行开始读入，直到匹配某行只包含 EOF 标记，将这块内容（长

串字符串）送给 :perl 命令作为参数。换用其他标记的理由，一般是内容本身包含 EOF 避免误解。

不过良好的实践，不推荐将 perl << EOF 裸写在某个 *.vim 脚本文件中，而应该封装在一个 VimL 函数中，最好再用 if has 判断保护，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 不能缩进，只能顶格写，即整行只能有 EOF 才表示 here 文档结束。这样封装之后，更能提高代码的健壮性与兼容性。然后就可按普通 VimL 函数一样调用了 :call PerlFunc() 。

当然，每次都写 if has 判断可能有点繁琐，那么可以将这个判断保护提升到更大的范围内，如：

```
if has('perl')

function! PerlFunc1()
    perl code;
endfunction

function! PerlFunc2()
    perl code;
endfunction

endif
```

或者将所有利用到语言接口的代码收集到一个脚本，然后在最开始判断：

```
if !has('perl')
    finish
endif
```

在 if_lua 或 if_python 接口中，还提供执行整个独立的 *.lua 或 *.py 脚本文件的命令，如下：

```
:luafile script.lua
:pyfile script.py
```

但是比较奇怪，if_perl 并没有类似的 :perlfile 命令，要实现类似功能，可能用 :perl require "script.pl" 命令，并且要注意 perl 的模块搜索路径问题。而在 :luafile 或 :pyfile 命令中，查寻命令行中提供的脚本文件，还是 vim 的工作， 取决于 vim 的搜索路径。

另外一个很有用的命令是 :perldo ， 它会遍历指定当前 buffer 范围的每一行（默认是 1,$ ），将 perl 的默认变量 $_ 设为遍历到的那行文本（不包括回车换行符），如果 :perldo 命令参数的代码段修改了 $_ ，它就会替换“当前”行文本。例如：

```
:perldo s/regexp/replace/g
:%s/regexp/replace/g
```

上面两行语句其实是一样的意义，都是执行全文正则替换，只不过第一行 :perldo 采 用 perl 风格的正则语法，它实际执行的是 perl 语句；第二行 :%s 就是执行 VimL 自己的正则替换。如果你想体会 perl 正则与 VimL 正则有什么异同，或对 perl 正则比 较熟悉，觉得某些情况下用 perl 正则更舒服，就可以用 :perldo s 代替 %s 试试 。

当然，:perldo 所能做的事情远不只 s 替换，s 在 perl 语言中只是一个操作符。perl 语言的单行语句非常强大，尤其是支持后置 if/for/while 的条件判断或循环，这就取决于用户的 perl 语言造诣了。

不过 :perldo 命令，与上一节介绍的过滤器机制略有不同，尝试用它实现给文本行编号的功能，最初的想法可能是：

```
:perldo $_ = "$. $_"
```

但这不能达到要求，$. 在 :perldo 遍历的每一行中都输出 0 ，这说明 perl 并 没有把文本行当前标准输入（或其他输入文件）处理，并没有给 $. 变量自动赋值。改 成如下语句能达到编号需求：

```
:perldo $_ = ++$i . " $_"
```

看起来有点像 perl 的黑魔法，其实不过是借助了一个变量 $i ，未定义变量当作数字用时被初始化 0 ，然后也支持像 C 语言的前置 ++i 语法，然后又将该数字通过点号 . 与一个字符串连接，代表行号的数字自动转化为字符串。这样创建使用的 $i 将是 perl 的全局变量，在执行完这条语句后，可以再用如下语句：

```
:perl print $i
```

查看 $i 的值，可见它仍保留着最后累加到的行号值。如果再次执行上面的 :perldo 语句对文本行编号，那起始编号就不对了。需要手动 :perl $i = 0 重置编号。但这也 正意味着，如果要求编号从任意值开始，上述 :perldo 语句就很容易适应。

在 lua 或 python 语言接口中，也有类似 :perldo 的命令。但是它们没有类似 $_ 默认变量的机制，:luado 与 :pydo 实际是在循环中为每行隐含调用一个函数，传入 line 与 linenr 参数代表“当前”行文本与行号，然后在参数的代码段中可以利用这两个参数进行操作，并可用 return 返回一个字符串，取代“当前”行。在写法上没 perl 那么简洁，而且在单行语句中不像函数的地方使用 return 也多少有点违和与出戏感。

9.2.4 目标语言访问 VIM

显然，如果使用一种语言接口，只是换一门语言自嗨诸如打印 Hello world 这种是没 有前途的。决定使用一种语言接口时，总是期望能利用那种语言更强大的能力，如更快的

运算速率或更丰富的标准库三方库功能，完成一系列数据与业务逻辑处理后，最终还是要通过某种形式反馈到 vim ，对 vim 有所影响才是。

为此，if_lua 与 if_python 都提供了专门的 vim 模块，在目标语言中将 vim 视 为一个逻辑对象，可从那语言代码中直接访问、控制 vim ，如设置 vim 内 buffer 的 文本，执行 vim 的 Ex 命令等。if_perl 也提供类似的模块，名叫 VIM，使用语法 与常规点号调用方法不同而已，perl 使用 :: 与 -> 符号。

以 if_perl 为为例，其 VIM 模块提供了如下实用接口：

- VIM::DoCommand({cmd}) 从 perl 代码中执行 vim 的 Ex 命令；
- VIM::SetOption({arg}) 设置 vim 的选项，相当于执行 :set 命令；
- VIM::Msg({msg}, {group}?) 显示消息，相当于 :echo ，但可以指定高亮颜色；
- VIM::Eval({expr}) 在 perl 代码中计算一个 vim 的表达式；
- VIM::Buffers([{bn}...]) 返回 vim 的 buffer 列表或个数；
- VIM::Windows([{wn}...]) 返回 vim 的窗口表表或个数。

其中，前三个接口方法只是执行 vim 的命令，perl 代码中不再关注其返回值。后三个方法是计算与 vim 相关的表达式，需要获得并利用其返回值。而 perl 语言的表达式是有上下文件语境的概念的。

VIM::Eval() 方法在标量环境中获得一个 vim 表达式的值，并转化为 perl 的一个标 量值。所谓 vim 表达式，比如 @x 表示 vim 寄存器 x 的内容，&x 表示 vim 的 x 的选项值。当然简单的 1+2 也是 vim 的表达式，但这种平凡的表达式直接在 perl 代码中求值也是一样的意义，没必要使用 VIM::Eval() 了。Vim 中的环境变量 $X 也与 perl 中 $ENV{X} 等值。 perl 的标量值具体地讲就是数字或字符串。但如果该方法在列表语境中求值，则结果也是一个列表，特别地是二元列表：

```
($success, $value) = VIM::Eval(...);
@result = VIM::Eval(...);
if($result[0]) { make_use_of $result[1] };
```

返回结果的第一个值表示 Eval 求值是否成功，毕竟参数给定的 vim 表达式有可能非法，如果成功，第二值才是实际可靠的求值结果。如果确信求值有意义，可直接用标量变量接收 VIM::Eval() 的返回值，那就是求值结果，可简化写法，省略成功与否的判断。

VIM::Buffers() 与 VIM::Windows() 的上下文语境就更易理解了，它符合 perl 的 上下文习惯：本来是数组的变量，在标量上下文表示数组的大小。所以不带参数的 VIM::Buffers() 返回所有 buffer 的列表，或在标量语境下返回 buffer 数量。如果 提供参数（可以一个或多个），就根据参数筛选 buffer 列表。如果想获取某个特定的 buffer，也得通过在列表结果中取索引，例如：

```
$mybuf = (VIM::Buffers('file.name'))[0]
```

你得保证 file.name 至少匹配一个 buffer，否则返回空列表，再对空列表取索引 [0] 是未定义的值。而且一般建议参数给精确，能且只能匹配一个 buffer ，否则如果匹配多 个，按 vim 的 bufname() 函数的行为，在歧义时也返回空。如果给的参数是表示 buffer 编号的数字，一般能保证唯一，只要是有效的 buffer 编号。给这个方法传多个参数时，就返回相应参数个数的 buffer 列表，例如：

```
@buf = VIM::Buffers(1, 3, 4, 'file.name', 'file2.name')
```

就将取得一系列指定的 buffer 对象，存入于 `@buf` 数组中。

一旦获得 buffer 对象，就可以用对象的方法，操作它所代表的相应的 vim buffer：

- `Buffer->Name()` 获得 buffer 的文件名；
- `Buffer->Number()` 获得 buffer 编号；
- `Buffer->Count()` 获得 buffer 的文本行数；
- `Buffer->Get({lnum}, {lnum}?, ...)` 获取 buffer 内的一行或多行文本；
- `Buffer->Delete({lnum}, {lnum}?)` 删除一行或一个范围内的所有行；
- `Buffer->Append({lnum}, {line}, {line}?, ...)` 添加一行多多行文本；
- `Buffer->Set({lnum}, {line}, {line}?, ...)` 替换一行或多行文本；

Window 对象也有自己的方法，请查阅相应文档，这里就不再罗列了。此外，还提供两个全局变量用于操作当前 buffer 与当前窗口：

- `$main::curbuf` 表示当前 buffer ；
- `$main::curwin` 表示当前窗口。

由于 `:perl` 命令执行的 perl 代码，就默认在 `main` 的命名空间（包）内，所以一般情况下可简写为 `$curbuf` 与 `$curwin` 。

第九章 VimL 混合编程

9.3* Perl 语言接口开发

本节将专门讲一讲 `if_perl` 接口的开发指导与实践经验，虽然只讲 perl ，但其基本 思路对于其他语言接口也可互为参照。

9.3.1 VimL 调用 perl 接口的基本流程

典型地，假如要使用（perl）语言接口实现某个较为复杂的功能或插件，其调用流程大概可归纳如下：

1. 定义快捷键映射，`nnoremap` ，这不一定必要，可能直接使用命令也方便；
2. 快捷键调用自定义命令，`command`；
3. vim 自定义命令调用 vim 自定义函数；
4. 在 vim 函数中使用 `:perl` 命令调用 perl 函数；
5. 在 perl 函数中实现业务运算，可能有更长的调用链或引入其他模块；
6. 在 perl 函数使用 VIM 模块将运算结果或其他效果反馈回 vim 。

在以上流程中，前三步是是纯 VimL 编程（细究起来，前两步准备动作还只是使用 vim ），第 5 步是纯 perl 编程，而第 4 步与第 6 步就是 VimL 与 perl 的接口过渡。接 口的使用只能按标准规定，打通一种可能，而要直接实现有意义的功能，重点还是回归到第 5 与第 3 步两门语言的掌握程度上。

整个流程是同步的，当 perl 代码执行完毕后，堆栈上溯，一直回到第 1 步的命令完成 ，才算一条 vim 的 `Ex` 全部完成，然后 vim 继续响应等待用户的按键。

但凡编程，要有作用域的意识，在这第 4 步中，首先是在 VimL 的函数的局部作用域中，首次进入的 perl 代码，是在 perl 的 main 命令空间。如果在 perl 的后续调用链 中，进入了其他命名空间，再想引用本次 vim 命令（第 2 步）或之前 vim 命令中在 perl main 命名空间定义的变量，就得显式加前缀 main:: 或简写 :: 也可。在 perl 代码中，使用 VIM 模块，只能直接影响 vim 的全局变量，它无法获知调用 :perl 命令所处的函数作用域或脚本作用域。如果有这个需求，请约定使用的全局变 :perl 代码同步返回时，及时从被影响的全局变量更新局部变量保存下来。

另一个基本意识是有关程序的输入输出。从 :perl 开始执行的代码，它的标准输出被重定向到 vim 的消息区。所以如果打印简单字符，:perl print 与 :echo 效果差不 多。在这里执行的 perl 不应试图从标准输入读取数据，如果需要输入，可以打开文件的方式（如临时文件，或确定的目标文件），或者利用 VIM 模块直接读取 buffer 内容。

9.3.2 Perl 代码与 VimL 代码解耦

虽然语言接口允许你将两种语言混用写在一起，但当真正想实现一些较复杂功能时，将两种语言的代码分别保存在独立的 *.vim 或 *.pl 是更好的代码维护与项目管理方式。而且也尽量将使用了 VIM 模块的 perl 脚本与未使用 VIM 模块的代码分开。

因为 VIM 模块只能是从 vim 执行的 perl 代码才可用。将那些未使用 VIM 模块的纯数据运算逻辑的 perl 代码独立开来，方便独立测试，也便于将其复用在非 vim 环境 下的常规 perl 脚本开发中。使用了 VIM 模块的 perl 代码，只方便在 vim 环境下测 试。如果一定要在外部独立测试调试，只能自己提供一个简易模拟版的 VIM.pm ，将在 脚本用到的 VIM:: 方法都实现出来（比如就打印调试信息之类）。

如下代码段可以判断 perl 是否运行在 vim 环境（是否通过 :perl 调用的）：

```
package main;
our $InsideVim = 0;
{
    eval { VIM::Eval(1); };
    $InsideVim = 1 unless $@;
}
```

perl 的 eval 语句块，有类似的 try ... catch 的功能，就是尝试执行 VIM 模 块的随便一个有效的方法，最简单就是 VIM::Eval(1) 了。如果不是从 vim 环境执行 ，eval 会出错，出错信息保存在 $@ 变量中。如果确实在 vim 环境中，eval 正 常执行，$@ 为空，unless 是条件取反，变量 $InsideVim 被置为 1 标记之。

然后就可以根据 $InsideVim 的值来做分支判断了。如果代码只设计在 vim 环境中使 用，当 $InsideVim 为假值时可直接 return 或 exit 。如果特意还是想在非 vim 环 境下通过测试，那就可以在 $InsideVim 为假时引用自写的简易调试版 VIM.pm 。

只为调试用的模拟 VIM 模块大致结构可以如下：

```
# File: VIM.pm
package VIM;
```

```
sub DoCommand{
    my $cmd = shift;
    print "Will do Vim Ex Command: $cmd\n";
}

sub Eval{
    my $expr = shift;
    print "Will eval Vim expression: $expr\n";
    return $expr;
}
```

也许还应该为 Eval() 函数添加自适应列表环境与标量环境的返回值，还有 Buffer 与 Window 对象的方法，模拟实现都会更复杂。故没必要求全，只根据实际情况，待测试的脚本用到哪些方法，首先让脚本能编译能运行，再考虑进一步模拟精度的必要性。当然最可靠的还是在 vim 中整合起来测试效果，只是在 vim 只能交互地手动测试，有时略有不便。

顺便提一下，使用 if_perl 时，不必显式声明 use VIM; 就能在相关代码中使用 VIM 模块。但使用 if_python ，还是要显式声明 import vim 的。

9.3.3 Perl 与 VimL 数据交换的几种方式

首先，简单的 perl 代码，如果 print 至标准输出的，在被 vim 调用时是打印到消息区的，因而可以用重定向消息的方法，将 perl 的标准输出内容捕获至 vim 变量中。例如 ，专门写个 ifperl.vim 存些基本工具函数，如：

```
" 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
```

这个函数将封装执行一段 perl 代码，将其标准输出当作一个变量返回（为简明起见，省略了错误等特殊情况处理）。一般更推荐调用 perl 函数，如此利用 s:execute() 也 很容易封装函数调用：

```
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
```

实际上，在 vim 命令行向 perl 函数传参数还得注意引号问题，这里也从略。然后，模拟 :pyfile 实现并未内置支持的 :perlfile 功能，也可简单封装成一个函数，如果也想关注执行一个 *.pl 可能的输出，可以改用上面的 s:execute() 函数：

```
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 与 use 语句有区别，各有用途。但都涉及搜索路径，在程序中推荐用 use lib 动态添加。可以将用于 vim 调用的 perl 脚本收集在一个目录（或专门的插件目录），并用 use lib 添加这个目录，便于 vim 使用。

其次，如果要用到的 perl 脚本，主要是一些工具函数，要利用其返回值的，而不是打印到标准输出的。这种情况下，若强行在 perl 处加一层打印函数，在 vim 处重定向消息，那是比较低效也不优雅的。另一个可考虑的替代的办法是专门设计几个全局变量槽让 perl 访问。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 中的三种数据类型，标量、列表、散列，分别可对应 VimL 变量的字符串、列表与字典，并且字符串在可能的情况下都可当作数字使用。当 perl 里的数据需要发往 VimL 时，临时借助事先规定好的这几个全局变量做缓存，只多调用一层转接函数，不影响原来 perl 函数的使用方式。

最后，其实要考虑的问题，是否真有必要将 perl 数据发还 VimL 。在协作完成一个功能时，得盘算好哪部分必须在 VimL 处完成，哪部分可集中在 perl 处完成，没必要的中间结果就别传回 VimL 处理了。

如果真要从 perl 频繁传出大量文本，自己用变量接收也不如用 VIM 内部的 Buffer 方法有效率。例如，也专门设计一个 buffer，取名 IFPERL.buf ，在 perl 中将需要查看的文本直接附加到这个 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(), @_);
}
```

这里直接将 ToVimBuffer() 函数的参数全部传给 Append() ，便支持同时添加多行（字符串列表）或一行（标量字符串）至 vim buffer 中。须提醒的是 Append() 方法的第一参数，不能使用 '$' 表示最后一行，只能是数字，因为这是在 perl 代码中， '$' 没有特殊行号意义，当作普通字符串转化为数字时，就是 0 ，结果就会添加到 buffer 最前面而不是最后面。

这种策略也适于记录被 vim 调用的 perl 代码执行过程的日志，直接发到某个 vim buffer 中查看。在开发调试时有奇效，比写日志文件更有效，然后由用户再决定有无必要保存日志。当然，完整的日志功能需要更灵活的控制，如在生产中就应该关闭，不打扰原则。

9.3.4 小结

使用 if_perl 接口混合编程的一个实用示例可参考这个插件： useperl 。本节上述引用的代码段也多是从该 插件简化而来的。该插件目前主要利用了 if_perl 实现 perl 语言编写补全，理论上 利用 vim 内嵌的 perl 解释器可达到语义理解级别，只是在具体实现细节，可能不甚完善。

然后说明一个事实，vim 支持多种语言接口，直接原因并非 VimL 本身设计多厉害（vim 的厉害之处更在其他整体综合上），而是因为那些脚本语言设计良好，方便嵌入其他程序。例如，perl 与 python 都可提供 C/C++ 扩展，而 vim 就是个 C 语言写的应用程序；还有 lua 语言最初设计目的就是便于嵌入到其他更大型的程序或服务上。所以 vim 利用 这些脚本语言的开发接口，编入它们的解释器，原非大惊小怪。也许，vim 还正想借这些 语言弥补自 VimL 脚本的不足。

那么，有了这些语言接口，是否就弱化了 VimL 脚本的意义了呢。那也不尽然，有些功能还是适合用 VimL 来实现，尤其是涉及用户界面接口部分，如快捷键 noremap 与自定 义命令 command 还有 GUI 版本的菜单 menu 。此外，VimL 兼容与移植性更好，毕竟其他语言接口不是默认编译选项。使用统一的官方 VimL 语言更有利于用户的交流与融合。

所以，随着 vim 的进化与发展， VimL 语言也应该稳步发展，这将成为 vim 文化与社区 不可或缺的一部分。

第十章 Vim 插件管理与开发

10.1 典型插件的目录规范

学 VimL 脚本的终极目标是写插件按需扩展 vim 的功能。在开始着手写插件之前，有必要先了解一下典型的、功能较齐全的插件，应该如何组织目录结构，按 vim 的习惯将不 同类别的功能放在相应的子目录下。

10.1.1 vim 运行时目录

插件的目录，可参考 vim 本身安装的运行时目录。所谓运行时目录，顾名思义，就是在 vim 运行时如果要加载 *.vim 脚本，应该到哪里找文件。

有两个相关的环境变量，可用如下命令查看：

```
:echo $VIM
:echo $VIMRUNTIME
```

如果从源码安装 vim ，且自定义安装于家目录的话，它们的值大概如下：

```
$VIM =  ~/share/vim
$VIMRUNTIME = ~/share/vim/vim81
```

所以 $VIM 指的是 vim 安装目录，而且不同版本的 vim 都将安装在该目录下，$VIMRUNTIME 就是具体当前运行的 vim 版本的安装目录。不过此安装目录不包括 vim 程序本身（那是被安装到 ~/bin 中的），主要是 vim 运行时所需的大量 *.vim 脚 本，相当于“官方插件”。该目录有哪些文件目录，可用如下命令显示：

```
:!ls -F $VIMRUNTIME
```

就是 shell 的 ls 命令，选项 -F 只是在子目录后面添加 / ，使得容易区分子目录与文件。也许直接从 shell 执行 ls 是被 alias 定义的别名，自动加上了一些常用选项，但从 vim 内用 ! 调用是不读别名的。

$VIMRUNTIME 既是官方目录，显然是不建议用户在其内修改或增删的。如果不是自定义安装在个人家目录，使用系统默认安装的 vim 的话，普通用户也无权修改。

于是 vim 提供了一个选项叫 &runtimepath （常简称 &rtp），那是类似系统 shell 的环境变量 $PATH，就是一组目录，只不过不用冒号分隔，而是用逗号分隔。可用如下命令查看 &rtp ：

```
:echo &rtp
:echo split(&rtp, ',')
```

通常，~/.vim/ 目录会在 &rtp 列表中，而且往往是第一个。另外，官方目录 $VIMRUNTIME 也在 &rtp 列表较后一个位置。当 vim 在运行时需要加载脚本时，就会依次从 &rtp 列表中每个目录（及其子目录）中查找，有时查找第一个就会停止。所以 $VIMRUNTIME 目录并不特殊，只是 &rtp 中一个优先级并不高的目录。对用户来说，~/.vim/ 目录才更特殊些，常被称为 vim 的用户目录。

一般建议用户将个人的 vimrc 及其他 vim 脚本放在 ~/.vim/ 目录中。可以用这个命令：

```
:echo $MYVIMRC
```

查看当前你运行的 vim 启动时读取 vimrc 。如果显示是 ~/.vimrc ，则建议将其移至 ~/.vim/vimrc 或软链接指向它。vim 会尝试读取 vimrc 的几个位置及顺序，也可用如下命令查看：

```
:version
```

然后提一下，如果是 windows 操作系统，没有 ~/.vim/ 目录。但它肯定有 $VIM 安 装目录，然后用户目录就是 $VIM/vimfiles 。

当了解了用户目录 ~/.vim/ ，就可以参照官方目录 $VIMRUNTIME 来组织管理自己的 vim 个性化配置及扩展脚本（插件）。

10.1.2 全局插件目录 plugin/

最简单的插件就是将 *.vim 脚本存到（某个） &rtp 的 plugin/ 子目录下。当 vim 启动时，就会读取（每个） &rtp 的 plugin/ 子目录下的 *.vim 脚本并加载。因为它们总是被加载，故有时称为全局性插件。

一般 vimer 初学阶段，倾向于完善与丰富自己的配置 vimrc 。当 vimrc 文件越来 越大感觉不便维护时，可将部分功能拆成独立脚本放在 plugin/ 目录下，毕竟这个目 录下的脚本也是能初始加载的，与合在 vimrc 中没有太大区别。可以想象一下，常规 vimrc 配置大约有如下内容：

- 使用 set 设置的选项
- 使用 map 系统列定义的快捷键
- 使用 command 定义的命令
- 自动事件命令组 augroup
- 自定义函数
- 为 gVim 定义的菜单
- 其他

如果为以上某部分内容进行了重度自定义，譬如快捷键，对每键盘上每个按键都仔细自己规划了一遍，甚至需要一些简单函数以便支付快捷键功能；那么就可尝试将这部分抽出来，另存为名如 ~/.vim/plugin/myremap.vim 的脚本。极端点，可以将 vimrc 中每部 分功能都拆出来扔到 plugin/ 目录。而 vimrc 只需留下这两行：

```
set nocompatible
```

```
filetype plugin indent on
```

这就是网上曾流传的所谓“最简配置”。第一行设置为不兼容 vi 模式，意即开启 vim 的 扩展功能；第二行是打开文件类型检测。另外我还建议在 vimrc 中定义一个环境变量 $VIMHOME 保存用户目录：

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

这样，在之后的 vimrc 或其他脚本的代码中，引用 $VIMHOME 就更有通用性，尤其是在需要手动加载（:source）脚本时。

不管是从大 vimrc 拆出脚本，还是从头开始写某个功能脚本放在 plugin/ 目录，都 要注意全局插件的一些特性。

其一是某个 plugin/ 目录下的所有 *.vim 脚本加载顺序不能保证。因此每个脚本要相应独立完成某个或某类功能，避免引用其他兄弟脚本定义的全局变量。如有这需求，类似 $VIMHOME 环境变量，还是在 vimrc 中定义吧，保证最开始被执行到。

其次是 plugin/ 的所有脚本还包含其子目录，即更深层次下的 &rtp/plugin/**/*.vim 脚本也会被自动加载。利用这个特性，可以对该目录进一步组织管理，将相关门类功能的脚本再放入更恰当的子目录名。但也要避免这个特性滥用，太深层次目录搜索比较耗时，可能会影响 vim 的启动速度。故一般不建议在 plugin/ 下再建子目录，最多再建一层。

如果 plugin/ 中脚本太多，影响 vim 启动速度，应该将其移出 plugin/ 目录。可能的直觉错误是在 plugin/ 下建个 backup/ 子目录，把某些不想用但想备用的脚本扔进去，这不管用，藏不住的。可以把 *.vim 脚本后缀改为 *.vim.bak，这就不会 被 vim 启动加载了。更好的建议是建一个与 plugin/ 平级的 plugin.bak/ 子目录 ，因为文件后缀名对 vim 编辑是重要的。

顺便说一下，在 vim 启动时，也有命令行参数可以指示 vim 在启动时跳过加载 plugin/ 的脚本。但一般日常使用时不必考虑这种差别。

10.1.3 类型插件目录 ftplugin/

与全局插件相对应的，是局部，具体讲，是与某种文件类型相关的插件，只在打开对应类型的文件时才生效。

文件类型是 vim 的一个概念，每个编辑的文件，都有个独立的选项值 &filetype，这 就是该文件的类型。直观地看，文件名后缀代表着其类型。但本质上这不是同一个概念。vim 只是主要根据文件名后缀来判断一个文件类型，有时还根据文件的部分内容（如前几行）来判断文件类型，用户还可以用 set filetype= 来手动设置一个类型。一种文件类型也可以关联好几个后缀名，比如 cpp、hpp 都是 C++ 文件，文件类型都是 cpp， 同样情况还有 htm 与 html 后缀名的文件，都认为是 html 文件类型。

文件类型插件要生效，还得在 vimrc 中添加 filetype plugin on 这行配置，这一般也是推荐必须配置。然后在打开文件并成功检测到属于某种文件类型时，vim 就会加载 &rtp/ftplugin/{&ft}.vim 脚本。

例如，每当打开 *.cpp 或 *.hpp 文件时，vim 都认为它属于 cpp 文件类型，它就会加载 ~/.vim/ftplugin/cpp.vim 脚本，以其其他 &rtp 目录下的 ftplugin/cpp.vim 。实际上，vim 搜寻文件类型插件脚本时规则很宽松，还会尝试搜索 cpp_*.vim 脚本，甚至子目录 cpp/*.vim 下的脚本。这目的是允许在同一个 ftplugin/ 目录中为一种文件类型提供多个插件脚本，它们都会被加载运行。

相比于 plugin/ 目录中的插件脚本只会在 vim 启动时执行一次，ftplugin/ 则可能 在 vim 运行时重复执行多次。每打开相应类型的文件（准确地说是 &filetype 选项值 被设置时触发）就会再次搜索并执行所有 &rtp/ftplugin 中所有匹配类型的脚本。

因此为了避免无意义重复工作，在文件类型插件脚本中，只推荐写那些确实每个文件（buffer）都需要独立设置的工作，如：

- setlocal 设置局部选项值
- remap 系列命令加上 <buffer> 参数，只为当前文件定义快捷键
- command 自定义命令也加上 -buffer 参数
- let 命令只修改 b: 作用域的变量

此外，还可以在相应的脚本中，通过 VimL 语法来控制脚本的实际执行。比如，参考官方目录的 cpp 类型插件，使用 :e $VIMRUNTIME/ftplugin/cpp.vim 打开，内容如：

```
" Only do this when not done yet for this buffer
if exists("b:did_ftplugin")
  finish
endif

" in c.vim
" let b:did_ftplugin = 1

" Behaves just like C
runtime! ftplugin/c.vim ftplugin/c_*.vim ftplugin/c/*.vim
```

开始几行通过判断 b:did_ftplugin 变量的存在性来决定是否继续加载当前这个脚本，一般在加载当前脚本时会将该值设为 1 ，这是 vim 官方推荐的文件类型插件的标准头写法。注意如果每个类型插件都是这样写，那是排他的意义，那就是加载了其中第一个类型插件的脚本，就不会再加载其他（有这个保护头的）脚本。虽然 vim 的机制会继续搜索其他匹配的类型插件脚本，但 VimL 语句层面上控制了不会重复加载，而这种控制是用户可选的方案。

最后一行表示 cpp 类型“继承”加载所有 c 类型的插件脚本，这是符合 C++ 语言与 C 语言特定业务关系的。这样就可以将 C/C++ 相关的都只写在 c.vim 类型插件中， 避免重复代码。事实上，那个 b:did_ftplugin 变量就只在 c.vim 中定义，不能在 cpp.vim 前面先定义，否则执行到 c.vim 是会被跳过。

有时在类型插件脚本中，比如定义局部快捷键时，不可避免要到调用特定函数以便封装具体实现。这种函数显然也只应该随文件类型插件加载，没用到过该类型就没必要加载，但是与局部快捷键需要为每个新打开文件定义的情况不同，函数定义最好只定义一次，不必为每个新文件重复定义。

如果是自己写在 ~/.vim/ftplugin/{&ft}.vim 中，脚本大致结构可以如下：

```
if exists("b:dotvim_ftplugin")
  finish
endif
let b:dotvim_ftplugin = 1

"

if exists("s:dotvim_ftplugin")
  finish
endif
let s:dotvim_ftplugin = 1

"
```

注意这里开头使用 b:dotvim_ftplugin 变量控制，不同于官方习惯的统一的变量 b:did_ftplugin，主要是不想有排他性。也就是说自己只想在 ~/.vim 用户目录下额外加些设置，执行完后还想加载官方的（或安装在其他目录的第三方的）同类型插件。

同样地，也可以在用户目录中让一种文件类型继承加载另一种文件类型。但是 :runtime 命令太泛了，会搜索所有 &rtp 目录。我们自己明确知道另一个目标文件类型是哪个脚本，就直接用 :source 会更有效率，例如在 ~/.vim/ftplugin/cpp.vim 中：

```
source $VIMHOME/ftplugin/c.vim
```

当然了，按个人实际情况，很可能都不会写纯 C 代码，那就直接维护 cpp.vim 脚本好了，不必额外有个 c.vim 脚本。另外，也有可能不同的文件类型都有部分共同设置代码，那也可以提取出来放在独立的 ftplugin/language.vim 脚本中，然后在各个具体的文件类型插件脚本中都调用这个脚本：

```
source $VIMHOME/ftplugin/language.vim
```

这里假设没有哪种文件类型名恰好叫 language，不过若防意外，也可以故意取个比较特殊的名字，如 ftplugin/_common_.vim 。

10.1.4 文件类型其他相关目录

与文件类型相关的目录，不止 ftplugin/ 这一个。ftplugin/ 一般是通用目的的 VimL 代码，还有其他几个目录，是 vim 为了实现其他具体功能时所需读取的脚本，虽然它们也是 *.vim 后缀名的脚本，理论上也可以写任意 VimL 代码，但实践习惯上只为完成特定功能。

因本书的主旨是讲 VimL 的，所以对这些目录或文件只简单罗列介绍于下：

- syntax/ 定义文件类型的语法高亮规则，基于正则匹配的；
- compiler/ 定义相应语言的编译命令及错误格式
- indent/ 设定缩进规则
- filetype.vim 检测文件类型的规则，自动事件 filetypedetect
- indent.vim 设置自动缩进的事件

- ftplugin.vim 文件类型插件加载机制

如果阅读这些官方脚本的源码，就会发现 ftplugin.vim 等就是利用自动事件实现的。显然也可以自己在 vimrc 中用 autocmd 实现根据文件后缀名加载特定的相关脚本。但是由于这个需求如此常见，官方已经帮我们做好了，并且支持了大量你见过的与未见过的编程语言。

另外，类似全局插件功能的，除了 plugin/ 外，也还有其他几个约定目录。如 colors/ 就是定义配色主题的。这里就不一一介绍了。

10.1.5 自动加载目录 autoload/

autoload/ 是放自动加载脚本的目录，在第 5.5 节介绍自动加载函数时就已提及。不过由于它在现代 vim 中非常重要，故这里再单独列出。自动加载机制是顺应 vim 发展而 提出的，也是 VimL 脚本语言的一大进步，因为 autoload/ 就相当于 perl/python 等 脚本语言存放模块的搜索路径。自动事件（autocmd）是 vim 内置机制，用户无法过多 干涉，autoload/ 自动加载函数是自动事件的一个重要扩充，允许用户在 VimL 语言层 面对函数与脚本的自动加载作灵活的控制。

自动加载函数是名字中含有 # 的函数，如 part1#part2#final() ，其函数名代表着（某个 &rtp 目录下的） autoload/ 目录下的相对路径，如 autoload/part1/part2.vim 。基于这种对应关系，定义自动加载函数的脚本不必在 vim 启动时事先加载，可以在 vim 运行时直接调用，首次调用时就会从 &rtp 中找到 相应的脚本自动加载。当然这是按 &rtp 顺序找到的第一个自动加载脚本就采用，所以 ~/.vim/autoload 往往有最高的优先级。但最好避免这种潜在

关于自动加载函数的用法，请回顾复习第 5.5 节，这里不重复了。不过全局变量名也可以采用 # 的标记，如 g:part1#part2#varname ，只在取值时会触发自动加载。

一般在开发较大型插件时，应该将主要实现函数都放在 autoload/ 目录下，并且建议将插件名再建一层子目录，这样该插件使用的函数名都有相同的前缀，或可称为命名空间 。而在 plugin/ 与 ftplugin/ 目录中只写简单的用户界面如快捷键、命令定义。如此在一定程度上就相当是 vim 接口与 VimL 实现的分离，有利于大型插件的项目管理。

10.1.6 善后目录 after/

after/ 是个很有趣的目录，每个 &rtp 目录下的 after/ 子目录又是一个 &rtp 目录，被自动添加到原来常规的 &rtp 列表之末。该 after/ 目录的结构可以与其父目录或其他 &rtp 目录一样。如果你了解数学上“分形”这个概念，可作此类比理解，就是“部分与整体拥有相似的结构”。

如果使用 :echo &rtp 命令，很可能在回显消息的末尾看到如下两个目录：

```
$VIMRUNTIME/after
$VIMHOME/after
```

因为 after/ 是自动添加到 &rtp 列表末尾，而 vim 在搜索运行时脚本时按顺序搜索 &rtp ，所以 after/ 目录可以保证尽可能后地被搜索。这机制有什么用途呢？

运行时脚本有两类明显不同的搜索方式。一种是搜索第一个匹配的脚本就停止，如 autoload/ 目录下的脚本，如此排在 &rtp 前列的具体更高的优先级。另一种是始终

搜索所有 &rtp 目录，如 plugin/ 与 ftplugin/ ，如此排到 &rtp 末尾的脚本具有更高的优先级。

如果用户安装了许多插件，每个插件被安置在独立的 &rtp 目录中（详见下一节的插件管理），那么不同 &rtp 目录下的同名脚本，就有可能冲突。因此在本插件目录下另建 after/plugin/ 或 after/ftplugin/ 目录可以大概率保证本插件提供的功能不被覆盖。

但是，一般的插件，除非有特别理由，不建议添加 after/ 子目录。强行武断地排他，提升自己的优先级。最好尊重用户的意愿，保留用户目录 $VIMHOME/after/ 让用户自 己决定如何解决冲突，覆盖其他插件的影响。

同时，也不要故意为难 vim ，在 after/ 目录下继续递归地建立 after/ 目录。

10.1.7 文档目录 doc/

最后要介绍的文档目录。vim 提供了详尽的在线使用手册，或叫帮助文档。在使用过程中如有任何疑难杂症，都推荐使用 :help 尝试。如果英文水平有限的，可以下载一份中文翻译文档。但最好还是习惯英文原文文档，毕竟命令与函数名是没办法翻译成中文的，熟悉 vim 官方文档使用的术语，有助于更好使用 vim 。

官方文档放在 $VIMRUNTIME/doc 目录下，就是 txt 纯文本文档。不过有特殊的约定格式，尤其是表示超链接目标与跳转到超链接的表示法，其他语法颜色对于 vim 已是司 空见惯。

用户可以并且建议为自己开发的插件编写文档，放在自己的 $rtp/doc 目录下，然后用 :helptags 生成索引（需要指定 doc/ 目录作为参数），以便支持跳转，这样就纳入了 vim 的帮助文档系统。用不带参数的 :help 打开帮助系统首页，在末尾部分有一节名为 LOCAL ADDITIONS 的，就列出了本地帮助文档，也就是除 $VIMRUNTIME 以外的 其他 &rtp 目录下的 doc/*.txt 文档。

最后提一句，善用帮助文档是学习与使用 vim 的不二法门。看过的任何书籍或技术博客文章，都大概率看过就忘记的，包括你正在看的这一本，它们的价值在于领进门，帮忙建立个概念，在实际遇到问题时还知道个搜索关键字，或者是 :help 的主题参数。至于详细使用细节，都以 vim 帮助文档为准。

第十章 Vim 插件管理与开发

10.2 插件管理器插件介绍

10.2.1 插件管理的必要性

上一节介绍了 vim 用户目录（~/.vim，并推荐设为 $VIMHOME 环境变量）。这在自己写简单脚本是很方便的，按规范将不同性质与功能的脚本放在不同子目录。但这有个潜在的问题，源于你不能总是自己造轮子，且不论是否有能力造复杂的轮子。

这世界上多年以来有许多狂热的 vim 爱好者，开发了许多作秀的插件，应该善加选择然后下载安装使用。但是如果都安装到 ~/.vim 目录，那来源于不同插件的脚本就混在一起了，既可能造成脚本同名文件冲突，也不利于后续维护（升级或卸载）。

后来，vim 提供了一种 vimball 的安装方式。就是将一个插件的所有脚本打包成一个文件，其实也是符合 VimL 语法的脚本，直接用 vim 的 :source 命令，就会把“包”内的文件解压出来，放到 ~/.vim 目录下，并跟踪记录解压了哪些文件，哪个文件来源于哪个安装包，然后将来要升级替换或下载删除时便有迹可寻。但这仍不可避免来源于不同插件的脚本同名冲突，且将个人用户目录搞得混杂不堪，对有洁癖的 vim 用户尤其是程序员是不能忍受的。

再后来，随着 github 的流行，版本控制与代码仓库的概念深入人心，vim 的插件使用与 管理思想也发生了革命性的变化。其实原理也很简单，关键还是 &rtp ，那不是一个目 录，而是一组目录，除了官方 $VIMRUNTIME 与用户目录 $VIMHOME 外，还可以将任 意一个目录加入 &rtp 列表。因此，可以将每个来源于第三方的插件放在另外的一个独立目录，在该目录内也按 $VIMRUNTIME 目录规范按脚本功能分成 plugin/ 、ftplugin/ 等子目录，再将其添加到 &rtp 中。如此，在 vim 运行时也就能在需要时从这个目录读取第三方插件内的脚本，就如果安装（拷贝）到 $VIMHOME 下的效果一 样。只是现在每个插件都有着自己的独立目录，甚至可直接对应 github 仓库，升级维护 变得极其方便了。

在所谓的现代 vim 时代，“插件”这词一般就特指这种有独立目录并按 vim 规范组织子目 录的标准插件，插件内的各子目录的文件一起协作完成某个具体的扩展功能。而之前那个用户目录 $VIMHONE ，建议只保留给用户自己保存 vimrc 及其他想写的脚本，不再被安装任何第三方插件。在这之前，$VIMHOME 目录尤其是 plugin/ 子目录下的每个脚本，也许都被称为一个插件，为了区分，或可称之为“广义的插件”。而从现在开始，单说插件时，只指“狭义”的标准插件（目录）。

当安装插件变得容易时，安装的第三方插件就会越来越多，这时又诞生了另一个需求。那就是如何再管理所有这些第三方插件？本章的剩下内容就来介绍一些便利的插件管理工具及其管理思路。这些插件管理工具本身也是个第三方插件。

10.2.2 pathogen

首先介绍一款插件： https://github.com/tpope/vim-pathogen。 看其名字 pathogen，大约是“路径生成器”的意思，其主要工作就是管理 vim 的 &rtp 目录列表。

按 pathogen 的思路，是在 $VIMHOME 约定一个 bundle/ 子目录，然后将所有想用的第三方插件下载安装到该目录。对于托管在 github 上的插件，可以直接用 git clone 命令，例如就安装 pathogen 插件本身：

```
$ mkdir -p ~/.vim/bundle
$ cd ~/.vim/bundle
$ git clone https://github.com/tpope/vim-pathogen
```

这样，插件安装部分就完成了，可以如法炮制安装更多插件。然后要 vim 识别这些插件，让它们真正生效，还要在 vimrc 中加入如下两句：

```
set rtp+=$VIMHOME/bundle/vim-pathogen
execute pathogen#infect()
```

其中第一句是将 pathogen 本身的插件目录先加到 &rtp 列表，如此才能调用第二句的 pathogen#infect() 函数。很显然，这是个自动加载函数，它会找到 autoload/pathogen.vim

脚本，加载后就能正式调用该函数了。该函数的功能就是扫描 bundle/ 下的每个子目录，将它们都加入 &rtp 列表，就如同第一句显式加入那样； 只不过 pathogen 批量扫描，帮你做完剩余的事了。

事实上， pathogen 插件只有 autoload/pathogen.vim 这一个脚本是关键起作用的，如果将该文件安装（下载或拷贝）到 $VIMHOME 中，那就没必要第一句显式将 pathogen 加入 &rtp ，因为它已经能在 &rtp 中找到了。如果在 Linux 系统，若为 安全或洁癖原因，不想复制污染用户目录，则可用软链接代替：

```
$ cd ~/.vim/autoload
$ ln -s ../bundle/vim-pathogen/autoload/pathogen.vim pathogen.vim
```

所以 pathogen 插件本身不必安装在 bundle/ 目录中，bundle/ 只是它用来管理其他后续安装的第三方插件。如果不想混在个人用户目录中，pathogen 可以安装在任意合适的地方，只要在 vimrc 将其加入 &rtp 或如上例做个软链接。

10.2.3 vundle 与 vim-plug

pathogen 在管理 &rtp 方面简单、易用，且高效、少依赖。只有一个缺点，那就是还得手动下载每一个插件。如果连这步也想偷懒，那还有另一类插件管理工具可用，如下几个都支持自动下载插件：

- https://github.com/VundleVim/Vundle.vim
- https://github.com/junegunn/vim-plug
- https://github.com/Shougo/dein.vim

其中，Vundle 出现较早，自动安装的插件默认也放在 ~/.vim/bundel 目录，只不过需要在 vimrc 中用 :Plugin 命令指定要使用的插件。现在基本推荐使用 vim-plug 代 替 Vundle ，用法类似，只不过使用更短的 :Plug 命令，而且支持并行下载，所以首次安装插件的速度大大增快。dein.vim 管理插件则不提供命令，要求直接使用 dein#add() 函数，并且插件安装目录默认不放在 ~/.vim/bundle 了。

这里仅以 vim-plug 为例介绍其用法。它只有单脚本文件，官方建议安装到 ~/.vim/autoload/plug.vim 中。但正如 pathogen.vim 一样，你可以放到其他位置，只是首先在手动维护这个插件管理的插件本身的 &rtp 路径。然后在 vimrc 进行 类似如下的配置：

```
call plug#begin('~/.vim/bundle')
Plug 'author1/plugin-name1'
Plug 'author2/plugin-name2'
call plug#end()
```

显然，plug#bigin() 与 plug#end() 是来自 autoload/plug.vim 脚本的函数，用于该插件管理工具进行内部初始化等维护工作，其中 plugin#begin() 函数可指定插件安装目录。然后在这两个函数调用之间，使用 :Plug 命令配置每个需要使用的插件。参数格式 author1/plugin-name1 表示来源于 https://github.com/author1/plugin-name1 的插件。:Plug 还支持可选的其他参数 ，比如用于配置复杂插件下载后可能需要进行的编译命令，这就不展开了。

在 vim 启动时，vim-plug 会分析 vimrc 配置的插件列表，如果插件尚未安装，则会自动安装，并打开一个友好的窗口显示下载进度及其他管理命令。如果在 vim 运行时编 辑了 vimrc ，修改了插件列表，并不建议重新 :source $MYVIMRC

，而是可以手动 使用 :PlugInstall 命令安装插件。一般只有在修改了插件列表配置后首次启动 vim 时才会触发自动下载的动作，当然下载速度取决于个人的网络环境，不过由于它的并行下载，横向对比其他插件管理工具的下载速度要快。在安装完插件之后启动 vim 显然就几 乎不会影响启动过程了。当然需要更新已安装插件时，可用 :PlugUpdate 命令。

vim-plug 这类插件管理工具的最大优点时功能丰富，不仅维护插件的 &rtp 路径，还 集成了插件的下载安装。当插件来源于 github 时，使得插件安装过程对于用户极其方便。相比于 pathogen ，它不仅是替用户偷懒免去手动下载过程，更简化了用户移植个人 vim 配置。比如想将自己的 vim 配置环境从一台机器挪到另一台机器，那只要备份 vimrc 而已（或 ~/.vim 目录），而插件列表内置在 vimrc 中，可不必另外备份，在新机器上首次启动 vim 时自动安装。

10.2.4 Vim8 内置的 packadd

从 Vim8 版本始，也提供了自己的一套新的插件管理方案，取代曾经昙花一现的 vimball 安装方式。核心命令是 :packadd ，而方案的名词概念叫 package ，可用 :help 命令查看详细帮助文档。

Vim8 内置的插件管理在思想上更接近 pathogen，就连 pathogen 的作者也在该项目主页上推荐新用户使用 Vim8 的内置管理方案了。因为这更符合 Vim 一贯以来的 Unix 思维 ，集中做好一件事。从 Vim 的角度，插件管理就只要维护好每个插件的 &rtp 路径就 尽职了。至于插件是怎么来的，怎么下载安装的，是用 git clone 还是 svn co ， 或是手动下载再解压，再或是用户自己在本地写的…… Vim 全不管，它只要求你把插件放 到指定的目录，就如一开始规定得把（广义的插件）脚本放在 plugin/ 目录一样。

事实上，之前的 vimball 就是 Vim 曾经试图介入用户安装插件过程的一种尝试。但是 vimball 没有成功推广，仅管那方案有可取之处。所以 Vim8 汲取经验教训，package 方 案不再纠结用户安装插件的事了，用户爱怎么折腾就怎么折腾。

现在，就来具体地介绍 package 方案如做插件何管理的工作。如果将 pathogen 管理的 插件迁移到 Vim8 的 package ，可用如下命令：

```
$ mkdir -p ~/.vim/pack/bundle/start
$ mv ~/.vim/bundle/* ~/.vim/pack/bundle/start/  #   bundle
$ ls -s  ~/.vim/pack/bundle/start ~/.vim/bundle #
```

这里，~/.vim/pack 叫做一个 &packpath 。那也是 Vim8 新增的一个选项，意义与 &runtimepath 或 &path 类似，是一个以逗号分隔的目录列表。

我们将 packpath 译为包路径，其下的每个子目录叫做一个“包”（package），每个包下面可以有多个插件（plugin）。如果包内的插件设计为需要在 vim 启动时就加载，就将这类插件放在包下面的 start/ 子目录中，如果不想在 vim 启动时就加载，就将插件放在包下面的 opt/ 子目录中。此后，在 vim 运行时，可以用 :packadd 命令将放在包下面的 opt/ 类插件按需求再加载起来（直接在命令行手动输入 :packadd 命 令，或在 VimL 脚本中调用该命令）。

也就是说，在 vim 启动时，会搜索 ~/.vim/pack/*/start/* ，将找到的每个目录都认为是一个标准插件目录，加入 &rtp 列表并加载执行每个 plugin/*.vim

脚本。当使 用 :packadd plug-name 时，就搜索 ~/.vim/pack/*/opt/plug-name/ 目录，如果找 到，则其加入 &rtp 并加载其下 plugin/*.vim 脚本。然后，&packpath 也不一定 只有 ~/.vim/pack/ 一个目录，用户可另外设置或增加包路径。

Vim8 将插件分为 start/ 与 opt/ 两类，这是容易理解的。因为 vim 要优化启动速 度，允许用户决定哪些插件得在启动加载，哪些可稍后动态加载。那为何又要在这之上再加个包的概念呢。那估计是前瞻性考虑了，预计将来 vim 用户普遍会安装很多插件，于是可以进一步将某些相关插件放在一个包内，以便管理。

对用户来说，如何使用包呢？如果从 github 下载的插件，很容易对应，就将作者名作为包名，于是该作者的所有插件都归于一个包。不过对个人用户来说，极可能只会用到某个作者的一个插件，并不会对他的所有插件都感兴趣；况且作者本身也可能只会公布少数一到两个 vim 插件。这样，每个包下面的插件数量太少，又似乎失去了包的初衷意义，而且包下面深层次目录只有一个子目录，利用率低，不太“美观”。

于是另有一个思路是根据插件功能来划分包名。想探寻某个功能，找到了几个来自不同作者开发的插件，各有优劣与适用场景，或者就是要多个插件配合使用，那就可以将其归于一个包。例如，上面介绍的插件管理插件，出于研究目的，可以都将其下载了，统一放在名为 plug-manager 的包内：

```
$ 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
```

显然，这些插件应该归于 opt/ 类，不能在启动时加载。

事实上，对个人用户而言，始终建议将下载的第三方插件安装在 opt/ 子目录下。否则，启动时自动加载的 start/ 插件可能太分散，也不利于维护。自己写的插件，放在 start/ 相对来说更为稳妥可信，但为了统一，也建议就放 opt/ 。确定需要启动加载的，就在 vimrc 中显式地用 :packadd 列出来。或者可以将这样的一份插件列表单独放在 ~/.vim/plugin/loadpack.vim 脚本中：

```
packadd plugin-name1
packadd plugin-name2
...
```

可见，这样的一份插件列表，就很接近 vim-plug 管理工具要求的 :Plug 配置列表了 ，只是没有下载功能而已。也许将来，会有配合内置 package 的插件管理工具，用来增 补自动下载的功能，供用户多个选择。

只是 package 方案出炉略晚，很多用户已经习惯了 vim-plug 这类的插件管理方式，短时间内不会转到内置 package 来。但是之前 pathogen 的用户，强烈改用 package 包管 理机制。

第十章 Vim 插件管理与开发

10.3 插件开发流程指引

10.3.1 标准插件开发

写插件有两个目的，然后是自用，扩展或增强某个功能，或简化使用接口。然后是发布到网站上给大家分享。Vim 的官网可以上传插件压缩包，不过现在更流行 github 托管。如 果仅是给自己用，插件脚本可以写得随意点，一些函数的行为也可以只接自己选定的一种固定实现。

但如果有较强烈的分享意愿，则应该写得正式点，这是一些实践总结的建议：

- 遵守第 10.1 节介绍的目录规范。
- 除非是单脚本插件放在 `plugin/` 目录中，较大型的插件如有多个脚本，则将主要函数实现放在 `autoload/` 子目录中，并且以插件名作为脚本名，或以插件名再建个子目录，如此插件名就相当于自动加载函数的命名空间。不过单脚本插件由于具有自包含、无依赖特性，在某些情况下也是方便的。
- 给用户提供配置参数（全局变量）定制某些功能的途径，变量名要长，包含插件名前缀然后接具有自释义性的多个单词，用下划线 `-` 或 `#` 分隔。并提供文档或注释说明。
- 如果插件的主要功能是提供了大量快捷键映射，最好为每个键映射设计 `<plug>` 映射 ，这种映射名应该与配置变量一样要长，包含插件名前缀，名字要能反映快捷键想做的工作。
- 最好在 `doc/` 提供详尽的帮助文档，要符合 `help` 文档格式规范。在文档中要说明命令、快捷键等用法，及配置变量的意义。文档也应该随脚本更新。
- 如果发布在 github 上，要提供一个 `readme.md` 说明文档，除了功能简介，至少包含安装方法与安装命令，便于让用户直接复制到命令行或 `vimrc` 配置中。

插件配置变量

支持用户配置全局变量的代码一般具有如下形式，在用户未配置时设定合理的默认值：

```
if !exists('g:plugin_name_argument')
    let g:plugin_name_argument = s:default_argument_value
endif
```

如果要设置默认值的可配置全局变量数量众多，则可以将这三行代码封装成一个函数，让使用处精简成一行。设置默认值的函数示例如下：

```
function! s:optdef(argument, default)
    if !has_key(g:, a:argument)
        let g:{a:argument} = a:default
    end
endfunction
```

还可以将所有默认值存入 `s:` 作用域内的一个字典中，键名与全局变量名一致。这样还能进一步方便集中管理默认值及设置默认值。

然后向用户说明，哪些快捷键是必须在加载插件之前（在 vimrc 中）设定值的，哪些快捷键是可以在使用中修改即生效的。

很多插件还习惯用一个 g:plugin_name_loaded 变量，来指示禁用加载该插件，在 plugin/ 脚本的开始处写入如下代码：

```
if exists('g:plugin_name_loaded')
    finish
endif
```

虽然依 vim 的插件加载机制会读取到这个脚本，但依用户的变量配置，有可能跳过加载该脚本的剩余代码，不再对 vim 运行环境造成影响。

插件映射设计

为了允许用户自定义快捷键，一个简单的方法是使用 <mapleader> ，让用户可按其习惯使用不同的 mapleader 。另一个更复杂但完备的做法是设计 <plug> 映射。当前有许多优秀插件的映射名使用类似 <plug>(PlugNameMapName) 的形式，把映射名放在另一对小括号中，看起来像个函数名。如果要伪装成函数，还可以就这样定义：

```
nnoremap <plug>=PlugName#MapName() :call PlugName#MapName(default_argument)<CR>
```

理解 <plug> 映射的关键，就是把 <plug> 当作类似 <CR> 、<Esc> 这样表示的一个特殊字符好了，只是它特殊到根本不可能让用户从键盘中按出来。这样让 <plug> 作为插件映射的 mapleader 就不可能与普通映射冲突了。为了也避免与其他插件的映射相冲突，还在 <plug> 字符之后加上表示插件名的一长串字符以示区别。

为了直观类比，再想象一下 vip 这个键序表示什么意义？就是依次按下 v i p 三个键，它会选定当前段落（空行分隔）！假如要开发一个插件，扩展 vip 选段落的 功能（主要目的还应是操作段落），例如根据文件或上下文语境，段落有不同的含义，不一定是空行分隔呢。那么该快捷键映射显然不能直接覆盖重定义 vip ，否则用户 v 进行可视选择模式会存在困难。至少应该定义为 <mapleader>vip 。对大部分用户来说， mapleaer 就是反斜杠，于是按 \vip 就触发该插件智能选段落的功能。

但这还不够灵活，更专业的做法是用 <plug>vip ，明示它来源于插件映射。但 <plug> 映射不是给用户最终使用的接口，因为 <plug> 字符根本按不出来。所以要双重映射：

```
nnoremap <plug>vip :call PlugName#SelectParagraph()<CR>
nmap <maplead>vip <plug>vip
```

注意第二个只能用 map 命令，不能用 noremap 命令，因为它要求继续解析映射。以上两行的组合效果相当于是：

```
nnoremap <mapleader>vip :call PlugName#SelectParagraph()<CR>
```

那为何要多此一举？程序界有句俗话，很多麻烦的事情，多加一层便有奇效。vim 有个函 数 hasmapto() 可判断是否存在映射，在开发的插件若支持用户自己定义映射，就该像全局变量配置那样，判断用户自己是否自定义过该快捷键了，只在用户未自己定义时，才提供插件的默认映射。例如：

```
if !hasmapto('<plug>vip')
    nmap <maplead>vip <plug>vip
endif
```

所以，让映射（特别是非内置的插件映射）有个纯粹的名字会方便很多。若直接以键序如 vip 指代一个映射功能，显得很诡异，程序可读性也不高。既然 <plug> 映射主要是 作为名字指称之用，不是让用户直接按的，那它的名字就可以更长更具体些，也可以再加些修饰符号（只要不是空格，否则让 map 命令解析麻烦）例如：

```
nnoremap <plug>=PlugName#SelectParagraph() :call PlugName#SelectParagraph()<CR>
nmap <maplead>vip <plug>=PlugName#SelectParagraph()
```

当然 PlugName 要替换为实际为插件取的名字。至于是否要在前后加 = 与 () 则无关紧要，只是风格而已。常见的风格还有将左括号 (紧接 <plug> 之后，括起整个映射名。但 <plug> 字符必须在映射名最前面。

插件的功能实现最终一般会落实到函数中，所以将插件映射名对应实现的函数名也是良好的风格，方便代码管理。但由于函数可以写得更通用些，可以接受参数调整具体功能，而快捷键映射没有参数的概念，所以不能强求映射名与函数名一一对应，而应该为每个常用参数的函数调用分别定义映射。例如，想用 \Vip 实现与 \vip 不同的功能：

```
nnoremap <plug>(PlugName#SelectParagraphBig) :call PlugName#SelectParagraph('V')<CR>
nmap <maplead>vip <plug>(PlugName#SelectParagraphBig)
```

虽然在插件映射名中也可以加括号与参数表示键序以求与函数调用外观一致，但未必更直观，而且传入多个参数时要注意不能加空格。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>
```

从对比中可见，当用参数 (1, '$') 调用函数时，不如直接取名为 SelectAll 更简 洁易懂。

插件命令设计

插件映射 <plug> 的设计颇有些精妙，在早期的插件中推荐用得比较多。后来自定义命令 command 越来越强大，于是在映射之外，再给用户界面提供一套命令接口也是一个选择。

如果将前面的 <plug> 映射名，去掉 <plug> 前缀（对用户使用来说，也相当于改为 : 前缀）及其他符号，或命名许可再略加省略简化，那就摇身转变为了合适的自定义命令名。当然相应地 map 要改为 command 命令，并注意不同的参数用法。

使用命令作为函数的用户接口，很容易实现传入不同的参数。因此更适合于那些不是非常常用的功能，没必要分别设计 <plug> 映射，毕竟命名也是桩麻烦事。

为了使命令更易用，务必提供合适的补全方法。命令自带提示记忆功能，也是它优于映射的一大特点。命令定义比普通映射复杂些，但理解起来不比 <plug> 映射困难。

提供了命令及相关说明文档之后，记得友情提醒一下用户，让用户知道可以自行、任意为他自己常用的命令定义快捷键映射，并自行解决可能的快捷键冲突。当然最好也提供一份快捷键定义示例，让用户可以拷入 vimrc 。例如：

```
nnoremap \vip :PNSelectPargraph<CR>
nnoremap \Vip :PNSelectPargraphBig<CR>
```

而这两个命令的定义，是写在插件脚本中的，可以像这样：

```
command PNSelectPargraph call PlugName#SelectParagraph()
command PNSelectPargraphBig call PlugName#SelectParagraph('V')
```

对于这个 vip 的例子，最后再提一句。直接将其定义为普通模式的快捷键不算是好的设计，那应该是操作符后缀（operator-pending）模式映射，那样就不仅支持 vip ，还 同时支持类似 dip 与 cip 等快捷键。不过本章只专注讲插件总体设计，就不深入具体实现细节了。

10.3.2 自动加载插件重构

大量安装插件的新问题

由于插件管理工具的进化，安装插件变得容易了，一些狂热用户就很可能倾向于搜寻安装过量的插件，启动 vim 加载几十上百个插件，并且让运行时目录 &rtp 迅速膨胀。虽然没有明确的数据显示，vim 加载多少个插件才算“过多”，才会显著影响 vim 启动速度 以及运行时在 &rtp 中搜索脚本的速度，仅从“美学”的角度看，太长的 &rtp 就显得 笨拙，不够优雅了。

让我们直观地对比下其他脚本语言如 perl/python 的模块搜索路径，典型地一般就五、六个，不超过十个。然而 vim 若加载 100 个插件，每个插件按标准规范占据一个独立的 &rtp 目录，那运行时搜索路径就比常规脚本语言多一个数量级了。（虽然从 vim 使用 角度看，似乎包路径 &packpath 对应着常规脚本语言的模块搜索路径，但从 vim 运行 时观点看，搜索 VimL 脚本却是从 &rtp 路径中的）

而且 vim 插件的规模与质量参差不齐，除了几个著名的插件，大部分插件其实都是“简单”插件，也就是只有少量几个 *.vim 文件，甚至就是追求所谓的单文件插件。那么为了一两个脚本，建立一整套标准目录，似乎有点大材小用。

上节介绍的 dein.vim 插件管理工具也意识到了这个问题，所以它提出了一个“合并”插件的概念，以便压缩 &rtp 。其实合并插件思想也很简单，有点像回归 vimball 的意 味。只不过原来的 vimball 的是无差别地将所有插件“合并”到用户目录 $VIMHOME ， 如此粗暴地入侵用户私人空间，仅管有监控登记在案，那也是不足取的。

所以，更温和点方案是专门另建“虚拟插件”目录，按标准插件的目录规范组织子目录，然后将其他第三方“简单”插件的脚本文件复制到该目录的对应的子目录中（尤其是 plugin/ 内的脚本）。这就实现了合并插件，所有被合并的插件共享一个 &rtp 目录 。而那些著名的大型插件，显然是值得为其独立分配一个 &rtp 的。至于如何判定“简 单”插件，那又是另一个层面的管理策略了。然而如何为被合并的插件保持可更新，那也是另一个略麻烦的实现细节。

不过，类似 dein.vim 的插件管理工具实现的合并插件，有点像亡羊补牢的措施。作为插件开发者，可以从一开始就考虑这个问题。如何组织插件结构可使得插件可合并，易于

与其他插件共享 &rtp ？这里就提供一个以此目的的重构思路。

基于自动加载机制重构插件

仍以上述 vip 插件为例。首先我们为这个插件确定一个名字，不如简单点就叫 vip 吧，这插件名字也足够高大上有吸引力。如果按标准插件规范，这整个插件应该位于 $VIMHOME/bundle/vip 或 $VIMHOME/pack/bundle/opt/vip 。再假设这是从一个简单插件开始的，目前主要只有 plugin/vip.vim 这个脚本。

首先，我们将 plugin/vip.vim 脚本移动到 autoload/vip/ 目录下，并改名为 plugin.vim ：

```
$ mkdir -p autoload/vip
$ mv plugin/vip.vim autoload/vip/plugin.vim
```

然后，编辑原脚本但改名后的 autoload/vip/plugin.vim ，在其末尾增加一个 plugin#load() 函数，空函数即可，或返回 1 假装表示成功：

```
function! plugin#load()
    return 1
endfunction
```

现在有什么不同呢？假设原来的 vip/ 插件目录已被加入 &rtp 中。那么移动改名之前的 plugin/vip.vim 会在 vim 启动时加载，而移动改名后的 autoload/vip/plugin.vim 并不会启动加载。但是可以通过调用函数（手动在命令行输 入或在其他控制脚本中写）:call vip#plugin#load() 加载。这个函数名意途非常明 确，足够简明易懂。如此触发脚本加载后，原来 vip 插件的所有功能也就加载进 vim 了 ，其中的命令与快捷键映射也就能用了。

既然现在 vip 插件的加载可由用户 VimL 代码主动控制了。那就可以将 autoload/vip 这个子目录复制到其他任意 &rtp 中。当然不建议复制到 $VIMHOME 中。可以单独建 个目录用于“合并插件”，比如 $VIMHOME/packed :

```
$ cd ~/.vim
$ mkdir packed
$ cp -r bundle/vip/autoload/vip packed/autoload/
```

在 Linux 系统，也可以用软链接代替复制，只要注意以后所指目标不再随意改名：

```
$ ln -s ~/.vim/bundle/vip/autoload/vip ~/.vim/packed/autoload/vip
```

然后，在 vimrc 或其他可作为管理控制的脚本中，加入如下配置：

```
:set rtp+=$VIMHOME/packed
:call vip#plugin#load()
```

如果有其他插件要合并入 packed/ 目录，依法炮制即可。将要加载的“插件”调用其相应的 #load() 函数，就如那些插件管理工具配制的插件列表。

自己要开发新插件，也可以从开始按这套路来，都不必另建插件目录，只要在自己的 $VIMHOME/autoload 建个子目录，写个 plugin.vim 脚本，脚本内定义一个 #load() 函数。

但是，如果想分享自己的插件，如何兼容之前的“标准”插件呢。或者说，就这个被改装重构的 vip 插件，如何回到兼容旧版本呢？那也很简单，plugin/vip.vim 脚本文件被移 走了，再建个新的就是，但是只要如下一行代码：

```
" File: plugin/vip.vim
call vip#pligin#load()
```

这样就可以了，用户（或者利用某个插件管理工具）可以像标准插件一样安装。如果介意 &rtp 路径膨胀（或其插件管理工具能识别），只要将 autoload/vip 目录复制到用户自己选定的另一个合适的共享 &rtp 即可。

简单插件扩展开发

原来本意为单脚本的插件，如果后来需要扩充功能，以致代码量上升，感觉塞在一个文件不太方便时，按标准插件的规范建议，也是将函数拆出来放在 autoload/ 目录中。

而如果像这里重构的 vip 插件，本来就是将主体脚本放在于 autoload/vip/plugin.vim 中，在该目录中添加与 plugin.vim 文件同层级的“兄弟”脚本，那显然就更加自然了。事实上，更合理的做法正是将插件的具体功能实现分别拆出放在不同脚本中。例如将选择段落的功能放在 select.vim ，将插入段落的功能放在 insert.vim ，替换段落的功能放在 replace.vim 中。当然，如何对插件功能抽象，是另一个层面的设计问题，与具体的插件及其规模有关。也许这几个插件适合都放在一个名为 operate.vim 的脚本 ，又或许更复杂的功能适合继续建子目录。

这里的关键只是想强调，不要将具体的功能实现（函数）放在 plugin.vim 中。 plugin.vim 原则上只写有关用户界面操作接品的定义。如 command 定义的命令， map 系列定义的快捷键映射。而且，<plug> 插件映射的定义也最好不要暴露在 plugin.vim 脚本中，它们应该定义在相关实现脚本中。plugin.vim 脚本只定义用户映射，即<plug> 插件可出现在 map 命令的第二参数中，不可出现在第一参数中。

当插件功能丰富起来后，就要向用户提供一些（全局变量）配置参数了。然后这些变量参数配置在哪里也是值得考虑的事了。传统习惯中，是简单地让用户配置在 vimrc 中。 但可想而知，当安装了许多插件后，你的 vimrc 很可能有大量代码在配置插件了。此后若删减或更换了插件，vimrc 中随意添加的插件变量配置也要记得删除。否则留下无意义代码，降低 vim 启动速度，污染全局变量空间，虽然那程度或许不算严重，但想想总是不爽不美的事。

参考加载插件的 vip#plugin#load() 函数，我们也可以相应地设计一个加载配置的 vip#config#load() 函数。这就意味着还有个 autoload/vip/config.vim 脚本与 plugin.vim 脚本并列。在这个 config.vim 脚本中，只使用简单的 let 命令定义 插件可用的配置变量的默认值，外带一个可空的 #load() 函数。真正有意思的是，允 许并建议、鼓励用户在其私人目录中提供自己的配置脚本，如 $VIMHOME/autoload/vip/config.vim 。由于个人 $VIMHOME 目录一般在 &rtp 最 前面，这个脚本如果存在的话会优先调用，否则就调用（被合并的共享 &rtp 目录下） 插件的默认配置。虽然，这句配置加载的调用函数应该写在 plugin.vim 的开始处。于 是 plugin.vim 脚本的总体框架现在大约如下：

```
" File: vip/plugin.vim
```

```
call vip#config#load()

" map
" command       vip#

function! vip#plugin#load()
    return 1
endfunction
```

如果没有在当前目录提供默认的 config.vim ，或担心用户提供的 config.vim 脚本 忘了定义 vip#config#load() 函数，为避免报错，可以将 :call vip#config#load() 这句调用包在 try ... catch 中保护。

让用户将插件配置在独立的 config.vim 中显然只应该是建议性的，而不应是强制性的。如果用户在 vim 启动时始终要加载的插件，相关插件配置被分散到 autoload/ 目录下 各个 config.vim 小文件中，反而会降低 vim 启动速度，不如将这些插件配置集中放在一个大文件如 vimrc 中。事实上，用户将各插件的全局变量配置放在哪里，并无影响，只是开发者要注意到这个现象。

这个 plugin.vim 脚本的体量可以很少，加载速度可以很快。关键在于定义命令时，调用其他 # 函数实现功能，就能在首次调用命令时触发加载插件中其他相关脚本。而快捷键映射，也建议定义为对命令的调用。如果习惯于 <plug> 映射，则将 <plug> 映 射本身定义为对具体 # 函数的调用（需要随 plugin.vim 加载，不能像 # 函数自 动加载）

用户在配置自己的 config.vim 时，可以推荐先从插件目录复制默认 config.vim 到 个人目录，在那基础上调整自己的参数值。如果变量名取得好，并且有一定的注释，那该配置文件也自带文档功能。更进一步，更激进的点是，如果 plugin.vim 脚本也足够简明，只定义命令与映射的话，用户也可以像复制 config.vim 一样复制到个人目录 $VIMHOME 对应目录下，然后直接修改快捷键定义（的第一参数）！这比在配置中约定一个诸如 g:plugin_name_what_key_map 的全局变量更直接。

文件类型相关插件

现在再来考虑文件类型相关的插件，这种可能需要多次加载的“局部”插件，比只需要一次加载的“全局”插件会复杂点。

假设我们这个 vip 插件要支持 cpp 文件类型了，它认为对于 C++ 源文件来说，什么叫“段落”应该有它自己特殊的处理。原则仍然是将所有运行时脚本放在 autoload/vip 目 录下。与 plugin.vim 脚本相对应的，文件类型相关功能可以建个 ftplugin.vim 。 然后在该脚本中设计一些意途明显的函数，如 vip#ftplugin#onft()，或者若该插件只想支持少数几种文件类型（大部分情况如此），直接定义 vip#ftplugin#onCPP() 函数 。在该函数内的语句只设置局部选项与局部映射等，供每次打开相应文件类型的 buffer 是调用。而局部映射可能需要用到的支持函数，可直接在 ftplugin.vim 脚本中定义， 也能保证只加载一次。

然后，如果 vip 还想兼容标准插件目录，那就再建个 ftplugin/ 子目录，其中 cpp.vim 文件只需如下一行调用：

```
:call vip#ftplugin#onCPP() "  vip#ftplugin#onft('cpp')
```

如果该插件想合并入共享 &rtp 目录，则指导用户将这行语句附加到个用目录的 $VIMHOME/ftplugin/cpp.vim 中。一般而言，如果用户常用 cpp 文件类型，关注 cpp 文件编辑，就该在个人目录建立这个文件，总有些自己想调教的选项或快捷键可以写在这个脚本中进行定制。然后安装的其他能增强扩展 cpp 功能的插件，若都像 vip#ftplugin#onCPP() 这个显式地在此加行配置，那对 cpp 的影响一目了然，也很好地体现了个人目录脚本的主控性，还能方便切换启用或禁用某个插件对 cpp 文件类型的 影响。

于是，在首次打开某个 *.cpp 文件时，会触发 autoload/vip/ftplugin.vim 脚本的 加载。会保证此时 vip/plugin.vim 脚本已加载，最好在 ftplugin.vim 脚本开头也 加入一行加载插件的调用语句。于是该脚本大致结构如下：

```
" 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 默认不能识别的文件类型，这样就不够了。例如这个 vip 插件还想 自创一种新文件类型，不如也叫 vip 吧，认为如 *.vip 或 *.vip.txt 后缀名的文 件算是 vip 类型。因为不能识别，所以不会自动加载 ftplugin/vip.vim 脚本。文件 类型的检测是基于自动事件机制，因此可以直接在 vip/plugin.vim 脚本中用 :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
```

定义了这个事件检测后，再打开 *.vip 文件 vim 就会自动加载 &rtp/ftplugin/vip.vim 脚本，可在其中调用 :call vip#ftplugin#onVIP() ，就如支持标准类型 cpp 那样。 但是也可以直接在 :autocmd 事件定义中直接调用函数，没必要间接通过 ftplugin/vip.vim 标准文件类型插件脚本来调用。可改为如下：

```
"File: vip/plugin.vim
augroup VIP_FILETYPE
    autocmd!
    autocmd BufNewFile,BufRead *.vip,*.vip.txt call vip#ftplugin#onVIP()
augroup END
```

其实对于标准文件类型如 cpp 也可以通过类似定义事件调用 vip#ftplugin#onCPP() ，但是不要在 ftplugin/cpp.vim 对该函数同时调用了，否则重复调用浪费工作。

插件自创文件类型还有一种典型情形是，该插件有功能打开一个类似管理或信息窗口时，想将该特殊 buffer 设为一种新文件类型，便于定义局部快捷键或语法高亮着色等。这种 buffer 还经常是 nofile 类型，不与硬盘文件关联，也不存盘。这时就不适合用 autocmd 根据文件后缀名来检测文件类型了。但是，由于这种 buffer 窗口是完全在脚本控制下创建打开的，直接设定 &ft 就行了。例如，我们的 vip 插件还在某个情况下打开一个提示窗口，不妨将其文件类型设为 tip ，于是在创建这种特殊 buffer 的 代码处，直接多加两行：

```
"   tip buffer
setlocal filetype=tip
call vip#ftplugin#onTIP()
```

注意，当把 &filetype 设为 tip 时，vim 也会自动去所有 &rtp 搜索 ftplugin/tip.vim 脚本。你可以利用或避免这种特性，决定是否要加 setlocal 这 行。而 vip 本身这个插件，对 tip 窗口初始化的入口函数，也像其他标准文件类型一样，集中放在 vip/ftplugin.vim 中定义。

其他标准插件目录的考量

以上，在将 vip 插件重构的过程中，将传统标准插件的 plugin/ 与 ftplugin/ 子 目录移到 autoload/ 下以插件名命名的子目录中，通过将插件名作为一级命名空间，来实现插件的动态加载，可达到加速 vim 启动速度，精简合并共享 &rtp 的目的。这 几乎可以涵盖 95% 以上功能拓展型的 vim 插件。

当然也有些特殊目的的插件不适合于 autoload 重构，比如定制颜色主题的 colors/ ，还有语法定义的 syntax/ 。理论上来说，语法也是文件类型相关的插件，也可以类似地移入 autoload/vip/syntax.vim 文件，将为每种文件类型定义语法的 :syntax 语句封装为函数，并由 ftplugin.vim 的相应函数来调用。但可能会有可用性与兼容性的问题。除非是插件内自创的临时文件类型如 tip 需要简单高亮时，可以考虑直接写在 vip#ftplugin#onTIP() 函数中（或由这个函数调用他处定义的语法支持）。

此外，还有 doc/ 帮助文档。这对用户使用参考很重要，但对 vim 运行时不重要，因此不在重构范围内。就仍按标准独立插件提供文档吧，如果需要合并插件，也直接复制 doc/ 文档到共享 &rtp 目录，也是简单的。

最后，想说明的是，这里所讨论的“重构”，主要是指插件开发思想上的重构。对于现存写好插件，没太大必要如何折腾，除非有相关插件管理工具能较智能地判断简单插件而自动合并与维护。更关键的是对于今后开发新插件，否认大小，简单或复杂的插件，都可以按这思路与规范，尽量将主体脚本封装一个 autoload/ 子目录中，以求最大化地追求支持动态或自动加载，也为合并插件共享 &rtp 打开方便之门。

笔者有个自写插件的集合 https://github.com/lymslive/autoplug 就在以此思路写了一些符合自身的实用插件。并
:PI 短命令，用于简化手 动调用 `:call xxx#plugin#load()` 的加载插件操作。

10.3.3 小结

本单介绍了两种插件开发的范式，一是继承传统，一是展望未来。传统的标准插件，主要依靠 vim 内置固定的几种机制，在不同的时机去指定的目录搜寻加载脚本。而后一种自动加载插件，更准确地说是可控的动态加载插件，则主要利用了 VimL 的一种通用的自动加载函数机制，能让开发者向用户提供更灵活的插件加载方式与配置方式。

正像学习任一门编程语言一样，学习用 VimL 进行插件开发，更重要的也是实践。只不过 vim 一贯追求个性性，具体的插件开发可能没那么强的通用性，因而不适合作为本书的正文内容。或许，那应该是另一个故事。而对读者来说，那也才算正式的起航。