Vimscript编程指南

kenvifire

Published
with GitBook

Table of Contents

1. Introduction
2. 关于本书
3. i. 前言
4. ii. 鸣谢
5. 0. 准备工作
6. 1. 输出消息
7. 2. 设置项
8. 3. 基础映射
9. 4. Modal映射
10. 5. 严格映射
11. 6. 前导符号
12. 7. 编辑你的Vimrc文件
13. 8. 缩写
14. 9. 更多映射
15. 10. 训练你的手指
16. 11. 本地缓存选项和映射
17. 12. 自动命令
18. 13. 本地缓存的缩写
19. 14. 自动命令组
20. 15. 操作符待定映射
21. 16. 更多操作符待定映射
22. 17. 状态栏
23. 18. 响应式编码
24. 19. 变量
25. 20. 变量作用域
26. 21. 条件分支
27. 22. 比较
28. 23. 函数
29. 24. 函数参数
30. 25. 数字
31. 26. 字符串
32. 27. 字符串函数
33. 28. Execute
34. 29. Normal
35. 30. Execute Normal!
36. 31. 基础正则表达式
37. 32. Grep操作符案例一
38. 33. Grep操作符案例二
39. 34. Grep操作符案例三
40. 35. 列表
41. 36. 循环
42. 37. 字典
43. 38. Toggling
44. 39. 函数式编程
45. 40. 路径
46. 41. 创建一个完整的插件
47. 42. 黑暗时代的插件布局
48. 43. 新的希望：通过Pathogen来布局插件
49. 44. 检测文件类型
50. 45. 基础的语法高亮
51. 46. 高级语法高亮
52. 47. 语法高亮深入

53. 48. 基础的折叠
54. 49. 高级折叠
55. 50. 区块移动理论
56. 51. Potion区块移动
57. 52. 外部命令
58. 53. 自动加载
59. 54. 文档
60. 55. 分发
61. 56. 接下来是什么？

Vimscript编程指南

《Vimscript编程指南》是一本可以供Vim使用者来学习如何定制化Vim的书籍。

这本书不会讲怎么去使用Vim。在阅读本书之前，你最好能够比较熟练使用Vim进行文本编辑，并且知道“缓存”，“窗口”以及“插入模式”这些概念。

这本书大致分为三个部分：

- 第一部分涵盖了可以用在 `~/.vimrc` 文件里来快速简单定制化Vim的一些命令
- 第二部分从编程语言的角度深入讲解了Vimscript，涵盖了变量，比较以及函数等内容。
- 第三部分讲解如何创建一个可以供其他人使用的简单插件。

这本书一直都可以在这里被在线免费阅读。

你可以花$8在Leanpub上买一本电子书（PDF，epub，mobi）

你可以花$20在Lulu上买一本简装本。

你可以花$40再Lulu上买一本精装本。

这本书的源码都可以在BitBucket和GitHub上找到。如果你发现了一些错误或者想做些改进，你都可以直接发起请求，但是我会保留这本书的版权，所以你如果觉得不舒服的话，我是可以理解的。

目录

i. 前言 ii. 鸣谢

1. 准备工作
2. 输出消息
3. 设置项
4. 基础映射
5. 高级映射
6. 严格映射
7. 前导符号
8. 编辑你的Vimrc文件
9. 缩写
10. 更多映射
    i. 训练你的手指
    ii. 本地缓存选项和映射
    iii.

前言

编程人员把想法转换成文本。

文本转换成数字，这些数字最后转换成其他数字，最后才使得一切得以实现。

作为编程人员，我们使用文本编辑器把我们头脑中的想法拿出来，转换成一大段的文本，并称之为“程序”。全职的编程人员会花费他生命中成千上万个小时来和他的文本编辑器打交道，在这段时间里，他们主要做一下几件事情：

- 把原始的文本从大脑里输入到电脑上。
- 修正文本上的错误。
- 重新调整文本，通过另外的方式来解决问题。
- 通过文档来描述为什么一件事是通过某种特殊的方式来实现的。
- 和其他的编码人员来沟通所有的这些事情。

Vim在处理这些问题上非常强大，但是如果你不能够为你的工作习惯和你的手指来定制它的话，就没法完全发挥它的作用。这本书会给你介绍Vimscript，它是用来定制Vim的主要编程语言。你可以用它来把Vim打造的更加满足你个人的操作习惯，这样你接下来的人生就可以更加高效地使用Vim。

在这过程中，我也会提到一些严格意义上和Vimscript无关的内容，一般会是一些关于学习和如何更加高效的内容。如果你整天都把时间浪费在折腾你的编辑器，而不是工作上，那么Vimscript用的再好是没有任何用处的，你需要在两者之间找到一个平衡点。

这本书和很多其他的编程书籍不一样。除了会告诉你Vimscript如何工作之外，它还会指导你去敲一些命令然后看会是什么样的效果。

这本书里有时候在讲解一个问题的时候，首先会把你带到一个死胡同里，而不是直接告诉你解决问题的“正确”方式。大部分其他的数据都不会这样，顶多只是在告诉你答案之后才告诉你这是个棘手的问题。但是现实的问题往往不是这样的。你经常会很快写一大段Vimscript代码，然后突然就碰到一个很棘手的问题。本书通过这种方式，而不是直接略过这种情况的方式来让你能够熟练掌握Vimscript的一些奇特用法，从而不断突破自己。然后达到熟能生巧。

这本书的每个章节都专注于一个主题。虽然每个章节都很短，但是里面的信息量很大，所以不要只是略过它们。如果你想要充分学习这本书的内容，你就需要亲自敲完每一行命令。也许你已经是一个经验丰富的编程人员，你能够很熟练地阅读代码。即使是这样，也没有问题，相比较学一门平常的编程语言而言，同时学习Vim和Vimscript是一种很不同的体验。

你需要敲完每一行代码。

你需要完成所有的练习。

我有两个原因要求你完成上面两个要求。第一，Vimscript是一座古老的殿堂，它里面有充满灰尘的角落，也有着曲曲折折的走廊。一个简单地配置甚至可以改变整个语言的工作方式。通过完成每一章的命令和练习，你会很容易发现你的Vim配置里地问题，并且能够很快地去修复。

第二，Vimscript其实就是Vim。在Vim里保存一个文件，你会用到 `:write` （或者是 `:w` ）然后回车。通过Vimscript保存文件，你会用 `write` 。其实你从本书里所学到的Vimscript命令都是你日常编辑文本都会使用到的，但是只有你把它们都熟练练习起来才会对你有所帮助，而这是没法只通过阅读来进行提高的。

我希望这本书能够对你有所帮助。这本书并不会介绍Vimscript的全部内容。它只会让你学习通过Vimscript这门语言来打磨你的Vim，让它变得更加合适你的胃口，让你能够写一些简单的插件，能够阅读其他人的代码（通过 `:help` 命令的帮助）,能够避免踩一些常见的坑。

祝您好运！

鸣谢

首先，我要感谢Zed Shaw，他编写了《Learn Python the Hard Way》一书，并且免费提供出来。这本书的格式和书写风格都都是来自它。

我还要感谢以下的GitHub和Bitbucket用户，他们通过提交pull请求，指出错别字，提出issue等多种方式进行了贡献：

- aperiodic
- billturner
- chiphogg
- ciwchris
- cwarden
- dmedvinsky
- flatcap
- helixbass
- hoelzro
- jrib
- lheiskan
- lightningdb
- manojkumarm
- manojkumarm
- markscholtz
- marlun
- mattsacks
- Mr-Happy
- mrgrubb
- NagatoPain
- nathanaelkane
- nielsbom
- nvie
- Psycojoker
- riceissa
- rodnaph
- rramsden
- sedm0784
- sherrillmix
- tapichu
- ZyX-I

对于以上名单没有列出来的人，我表示诚挚的歉意。

预备工作

=======

为了能够学习本书的内容，你最好能够安装最新版的vim，在本书编写的时候，最新版本是7.3。新版本的书籍一般都会保持向后兼容，所以任何7.3以后的版本都是可以的。

其次，你应该能够使用vim编辑文本文件。

你应该了解Vim的一些基本术语，例如“缓冲区”，“窗口”，“一般模式”，“插入模式”以及“文本对象”。

如果你还没有达到上面的要求，你可以通过vimtutor来学习一下，然后使用并且只用vim一到两个月，直到你的手指能够适应vim，再来学习本书。

你还应该有一些编程的经验，如果你从来没有编写过程序，那么你可以先看看《Learn Python the Hard Way》，然后再来学习本文。

创建Vimrc文件

如果你已经知道vimrc文件了，并且系统里已经有了该文件，那么你可以直接跳到下一章了。

vimrc文件里包含一些vim脚本，每当vim启动的时候，它都先自动加载这个文件里的脚本。

在Linux和Max OS X系统里，vimrc文件在你的home目录下，并且被命名为.vimrc。

在windows下，这个文件在你的主目录下，并且被命名为_vimrc。

为了更方便的找出任何系统下的vim文件的路径，在vim里运行 `:echo $MYVIMRC` 。在屏幕的下方会显示vimrc的文件路径。

如果没有这个文件的话，那就创建一个。

输出消息

=======

我们看的第一个vim脚本命令是`echom`。

你可以通过`:help echom`来查看关于这个命令的完整手册。在你学习这本书的过程中，建议你都用`:help`命令来查看每个新学习到的命令，来加深你对命令的理解。 运行下面的命名： `:echo "Hello,world!"` 在你屏幕的下面，应该能看到“Hello,world!”这几个字符。

持久化消息

现在，运行下面的命令： `:echom "Hello again,world!"`

现在，你在屏幕的下方看到的应该是“Hello again,world!”

现在来，看看这两个命令的区别，首先运行一个新的命令： `:messages`

你应该可以看到一个消息列表，"Hello,world!”不会出现在这个列表里，但是“Hello again,world!”在里面。

当你在深入学习本书时，后面会写一些复杂的脚本，那么你将需要打印一些信息来帮助你调试程序。简单的`:echo`命令会可以用来输出信息，但是一旦脚本运行完毕，消息也就丢失了。但是用`:echom`来输出的话，会保存消息，并且后面可以用`:messages`来查看。

注释

在我们继续学习其他命令之前，我们需要了解一下如何进行注释。当你在写vim脚本的时候（在vimrc文件或者其他文件里），你可以用"字符来添加注释，例如：

```
"Make space more userful
nnoremap  za
```

但是这并不一总是有效（在Vim脚本的某些丑陋的地方是无效的)，但是大部分场景下是有效的，我们后面会讨论为什么在某些场景是无效的。

练习

=======

- 阅读`:help echo`的内容。
- 阅读`:help echom`的内容。
- 阅读`:help messages`的内容。
- 添加一行脚本到你的vimrc文件里，使得每当你打开vim的时候会显示>^.^<。

设置项

=======

Vim有很多配置项，通过修改配置项的值来改变vim的表现。

Vim里有两类配置项，一类是布尔值配置项（只能是“on”或者“off”），另一类是具体值的配置项，这种配置项有一个具体的值。

运行以下命令：

```
:set number
```

这时候你的Vim左边每行的开始的地方都会有一个行号，现在运行以下命令：

```
:set nonumber
```

这时候行号又消失了。number是一个布尔型的配置项——它可以被打开也可以被关闭。通过 `:set number` 命令可以打开它，通过 `:set nonumber` 可以关闭它。

切换配置项

=======

你可以切换配置项，即是把它的值设成和当前状态相反的值。运行下面的命令：

```
:set number!
```

行号又会出现了。现在继续运行下面的命令：

```
:set number!
```

行号又消失了，在布尔型配置项前加“!"可以对该项的当前值取反。

查询配置项

=======

你可以通过使用“?”来向Vim查询对应选项的值。运行下面的命令看看会出现什么： `:set number` `:set number?` `:set nonumber` `:set number?` 可以看到，第一条:set number?命令显示的是number，而第二条:set number?命令显示的结果则是:nonumber。

值类型的配置项

=======

有些配置项需要的是一个具体的值，而非只是被打开或者是关闭。运行下面的命令，并仔细看看每条命令运行后vim里的变化（译者注：主要看行号的宽度的变化）：

```
:set number
:set numberwidth=40
:set numberwidth=4
:set numberwidth
```

numberwidth选项的值决定了行号这一列的宽度。 你可以试试看看其他常用的选项的值:

```
:set warp?
```

```
:set shiftround?
:set matchtime?
```

一次设置多个值

=======

在一个set 命令里，一次可以设置多个值，试试以下命令：`:set number numberwidth=6`

练习

=======

- 阅读`:help 'number'`（注意单引号）的内容。
- 阅读`:help relativenumber`的内容。
- 阅读`:help numberwidth`的内容。
- 阅读`:help warp`的内容。
- 阅读`:help shiftround`的内容。
- 阅读`:help matchtime`的内容。
- 在你的vimrc文件里按照你喜欢的样式设置这些值。

基本映射

=======

Vim里唯一能够让它屈服你的意愿的功能就是按键映射。

键盘映射可以让你告诉Vim：“当我在按这个键的时候，我想做我想做的事情，而非正常情况按下该键所该所的”。

我们首先从一般模式下的键盘映射开始。后面的章节里我们会继续讨论插入模式和其他模式下的按键映射。

在Vim里打开一个文件，并输入几行文字，然后运行以下命令：

```
:map \ x
```

把光标移动到文本的任何位置，然后按“\”，你会发现光标下的字符会被删除了，就像按下了“x”一样。

Vim里已经有了专门用来删除当前光标下字符的快捷键，那我们现在就把“\”映射成一个更实用的功能。现在运行下面的命令：

```
:map \ dd
```

现在把光标移动到某一行文本上面，再次按“\”。这时候，光标所在行的整行都被删除了，这也即是`dd`命令的功能。

特殊字符

=======

你可以用来告诉vim来实用一些特殊的按键。试试以下命令：

```
:map <space> viw
```

把光标移动到一个单词上面去，然后按空格键。vim将会选中这个单词。

也可以通过以下方式来修饰需要按下Ctrl和Alt的快捷键。试试下面的命令：

```
:map <c-d> dd
```

现在按`Ctrl+d`的效果和`dd`的效果一样了。

注释

=======

还记得我们在第一节里讲到怎么注释的吗？按键映射是vim里注释不起作用的地方之一。试着运行下面的命令：

```
:map <space> viw "Use space to select a word
```

当你再次按下空格键时，一个奇怪的现象就会出现了。为什么呢？

当你按下空格键时，vim会认为你输入的命令是`vim <space>"<space>Use<space>space<space>to<space>select<space>a<space>word`。

很显然，这并不是我们想要的结果。

这个映射很有趣，因为如果你再仔细看看这个映射最终的结果，你会发现其中的原因。试着去找个究竟来，如果你实在找不出来的话，也没关系，

我们后面也会讲到这个问题的。

练习

=======

- 把‘-’映射成"删除当前行，并把它粘贴到下一行"的功能，这样就可以让你只用按一个键就可以实现把文本往下移的功能了。
- 把上面的映射加入到~/.vimrc文件里，那么你以后就可以随时使用这个功能了。
- 把‘_’映射成把当前行向上移一行的功能。
- 把上面的映射也加入到~/.vrimrc文件里。
- 猜猜看，如何才能取消一个按键的映射，让它恢复它原有的功能。

Modal Mapping

=======

上一章节里我们讲述了怎么在vim里建立按键映射。我们用map命令让normal模式下的按键能够进行映射。如果你在阅读本章之前，自己在把玩一番的话，你会发现键盘映射在visual模式下也是有用的。

我们可以通过nmap，vmap，和imap命令来对键盘映射所生效的模式进行指定。这些命令分别告诉vim，映射只在normal，visual或者是insert其中某一个模式下有效。

运行下面的命令：

```
:nmap \ dd
```

现在把光标移动到一行文本上，进入到normal模式，然后按'\'，vim会删除当前行。

现在进入visual模式，通样按'\'，什么效果也没有，这是因为nmap告诉vim这个映射只在normal模式下有效（并且在visual模式下'\'并没有任何效果）。

再运行下面的命令：

```
:vmap \ U
```

进入visual模式，并选中一些文本（英文文本），然后按'\'，vim会把这些文本（应为文本）都转成大写。

再分别在visual模式和normal模式下试试'\'键，你会发现它在两种模式下的功能是完全不一样的。

肌肉记忆

=======

似乎同一个按键在不同的模式下做不同的事情是一个很糟糕的主意，因为每次按键之前都得考虑一下当前处于什么模式。这样的话，不就会抵消原来映射所节省的时间么？

但实际上不会存在这样的问题的。当你熟练使用vim的时候，你不会再只是考虑你所要按的某个键，而是具体的操作。当要删除某一行的时候，你想的是“删除改行”，而非是“按下dd”。你的手指和大脑会记住你的键盘映射，而具体的按键则是你潜意识下的操作。

insert模式

=======

现在，我们已经讨论了visual和normal模式下的映射问题，现在我们继续讨论在insert模式下的映射。

运行下面的命令：

```
:imap <c-d> dd
```

也许你会认为当你在normal模式下按下'Ctrl+d'时，当前行会被删除掉。这个功能确实不错，因为你再也不用跑到normal模式下去删除文本行了！

试试吧。但是实际上它却不起作用，而且只是在你的文本里添加了两个‘d’！这个确实毫无用处。

事实上vim是在做你让它去做的事。我们在告诉它：“当我按Ctrl+d的时候，把它映射成按两个‘d’所做的事情”。

很好，事实上，当你在Insert模式下，并且按下‘d’两次时，就会在你的当前行输入两个‘d’！

为了让这个映射有效，我们需要说的更清楚点。用下面的命令来替换上面的那个映射：

```
:imap <c-d> <esc> dd
```

表示vim里的Escape键，按这个键就让我vim切换出insert模式。

现在再试试这个映射，它确实删除了当前行！但是你会发现你现在却是在normal模式下了。这个问题很明显了，因为你告诉vim离开insert模式，然后删除一行文本，但是没有告诉它再进入insert模式。

用下面的命令来修复上面的那个问题：

```
:imap <c-d> <esc> ddi
```

后面添加的i可以vim进入插入模式，最终这个映射完成了。

练习：

=======

- 建立这样一个映射：当你在插入模式下输入Ctrl+u时，会把当前的单词整个转成大写。在visual模式下，按下u可以让选择的文本变成大写。我觉得这个功能很有用，尤其是在输入一个很长的常量名的时候，我只需要输入这个名字的小写，然后用这个映射来把它变成大写。
- 把上面的映射加入到你的~/.vimrc文件里。
- 建立这样一个映射：当你在normal模式下时，按下Ctrl+u会把当前的单词整个转成大写。这个映射和上一个略有不同，因为你不需要进入normal模式，并且你最终是要停留在normal下，而非是insert模式。
- 把这个映射也加入到你的~/.vimrc文件里。

严格映射

=======

坚持住，事情会变得有点复杂的。

到现在为止，我们已经使用了map,nmap,vmap和imap来建立按键映射，用以节省时间。这个方法确实很有效，但是这个方法也存在问题。

运行下面的命令：`nmap - dd` `nap \ -`

现在在normal模式下，按下'\'试试，发生了什么呢？

当你按下'\'时，vim找到对应的映射，也即是相当于'-'。但是我们已经把'-'映射成了其他的功能！最终vim会根据'-'的映射来操作，也即是'dd'命令，所以它把当前行给删掉了。

当你用vim来建立映射时，vim会把映射的目标所对应的映射也考虑进来。这个功能看起来确实很不错，但是实际上是个很恶心的功能。我们后面会讨论这个问题。

递归映射

=======

运行下面的命令: `nmap dd O<esc>jddk`

你可能会认为这个命令会把dd映射成以下几步：

- 在当前行的上方插入一个新行
- 退出插入模式
- 往下移动一行
- 删除当前行
- 移动到刚刚创建的那个空行

但是实际上，这个映射的功能是删除当前行。试试

当你按下'dd'的时候，vim好想死掉了。这时候你按下Ctrl+c，vim会恢复运行，但是你的文件里会多了很多很多的空行！到底是怎么一回事呢？

- dd被映射了，所以执行映射
  - 插入一个新行
  - 退出插入模式
  - 移动到下一行
  - `dd` 被映射了，所以执行映射
    - 插入一个新行
    - 退出插入模式
    - 移动到下一行
    - dd被映射了，所以执行映射
    - .....

所以这样递归下去，这个映射是无法终止的。

副作用

=======

并不只是我们所学到的*map命令会有递归的风险，并且我们安装的其他的插件也会有这样的风险。

当你安装一个新的插件时，你可能不会使用也不会记住这个插件所使用的映射。如果你用了，那么你就需要你的~/.vimrc文件去看看你自己的映射按键没有被插件使用，同时还要保证插件使用的映射你自己没有使用。

这个问题会让插件安装比较麻烦，并且很容易出现问题。vim里肯定有更好的办法。

非递归映射

=======

vim提供了另外的一些建立映射的命令，它们不会进行递归的映射。试试下面的命令：

```
:nmap x dd
:nnoremap \ x
```

现在试试按'\'，看看会发生什么。

当你按下'\'时，vim会把他映射成x的操作，并且忽视x的进一步映射。这次不再是删除当前行了，只会删除当前的字符。

所有的*map*命令都有对应的noremap命令来忽视递归的映射，它们分别是:nnoremap,vnoremap和inoremap。

什么时候使用非递归映射

=======

什么时候需要用这些映射命令的非递归版本来替代递归版本呢？答案是：一直都需要。

注意，是一直都需要！

使用最基本的nmap命令会让你在按照插件或者建立新的映射时非常的痛苦。为了省去不必要的麻烦，建议你还是多输入几个字符，保证不要出现因递归的映射带来的负效应。

练习

============

- 把你的~/.vimrc文件里的所有的映射命令换成它们的非递归版本。

前导字符

==========

我们已经学会了如何建立按键映射来让我们用起来比较爽，但是你也许会发现另外的一个问题。

每次我们使用类似于 `:nnoremap <space> dd` 命令来对空格键进行映射时，我们就会覆盖空格键原有的功能，但是如果我们后面需要使用空格键原来的功能怎么办呢？

还有很多按键在vim里我们是很少使用的。例如'\'没有任何功能，-，H，L和空格键基本上都不会使用到。具体其他的不使用的键你自己可以根据自己的使用习惯来发掘。

这些不使用的键可以随意哪来映射，但是只有5个。但是vim神奇的自定义功能可以帮我们实现更多。

映射按键序列

==========

不像Emacs，vim可以很简单的映射多个按键，运行下面的命令：

```
:nnoremap -d dd
:nnoremap -c ddO
```

然后在normal模式下尝试输入'-d'和'-c'（速度要快）。这个命令的第一行是自定义了一个删除一行文本的功能，第二行定义了清除一行，并转换到插入模式的功能。

这样也就意味着我们可以用一个不使用的按键，例如'-'，来用作前导字符，然后再根据这个前导字符来建立相应的映射。这也就意味着我们需要多按一个键来触发映射，不过这个对于手指而言是很容易掌握的。

如果你认为这是个不错的注意的话，那就对了，vim确实也有前导字符的功能。

2、前导字符

==========

vim称前缀按键为"leader"。你可以任意设置你的leader键，运行下面的命令：

```
:let mapleader = "-"
```

你可以用任意键代替'-'，我个人比较喜欢用'，'，虽然它是一个很常用的标点，但是它很容易被按下，所以我比较喜欢用逗号。

当你用来定义键盘映射的时候，你可以不管lead键被设置成什么，vim会自动帮你映射的。运行下面命令：

```
:noremap <leader>d dd
```

现在试着按leader键加上d，vim会删除当前的文本行。

为什么要用呢？为什么不直接把前导按键放在映射命令里呢？

首先，你可能在后面的某些时候需要用你的前导按键原来的功能。把它定义在同一个地方方便后面修改。

其次，当别人在参考你的~/.vimrc文件的时候，他们马上就会知道键的意思，那么他们需要的话就可以直接拷贝你的映射到他的~/.vimrc文件里，而不需要修改对应的leader键。

最后，有些vim插件会根据你的leader键来做映射的，所以如果你设置了leader键的话，就可以直接用这些映射了。

本地**leader**键

==========

vim有另外一个leader键叫做本地leader键，本地的意思是这个前导键所对应的映射只会对某些类型的文件例如Python文件或者HTML文件有用。

我们后面会讨论怎么对特定的文件类型设定本地映射，不过在这里你可以先试着设置本地映射：

```
:let maplocalleader="\\"
```

注意我们需要用'\'而非'\'，因为'\'在字符串里是转义符。在后面会讲到这个的。

现在你可以在映射里像用leader键一样使用localleader键，它们只是映射的按键不同而已。

练习

==========

- 阅读:help mapleader
- 阅读:help maplocalleader
- 在你的~/.vimrc文件里设置mapleader 和maplocalleader键
- 把你的~/.vimrc里的所有快捷键都改成带leader键的映射，以免覆盖之前本来的功能。

编辑你的**vimrc**文件

==========

在我们继续学习vim脚本之前，我们先找到一个方便的方式来给我们的~/.vimrc文件添加映射。

当你在对一个问题进行猛烈编程时，这时你可能会发现一个新的快捷键让你的工作变得更轻松，然后你需要马上把它加到你的~/.vimrc文件里以免忘记了，但是你也不希望自己分心。

这个方法就是让你能够更容易找到让文本编辑变得更容易的方法。

编辑映射

==========

让我们添加一个映射，它实现这样的功能：在新窗口里打开我们的~/.vimrc文件，然后我们就可以编辑它，然后继续回到我们的编码工作上。运行下面的命令：

```
:nnoremap <leader>ev :vsplit $MYVIMRC<cr>
```

我把这个命令命令为“编辑我的vimrc文件”，映射为ev。（ev取“edit my vimrc file”中edit 和vimrc的首字母）。

$MYVIMRC是vim里的一个变量，它指向你的~/.vimrc文件。不用担心这个还没学到的变量，相信我，它可以用的。

vsplit在竖直方向分隔出一个新的窗口。如果你喜欢使用水平方向的窗口，你可以用split，不过在现在的电脑屏幕都这么宽，我很不理解你为什么要用竖直的。

现在花点时间在你的大脑里想想这个命令的功能：“在竖直方向的新窗口里打开我的~/.vimrc文件”。为什么会凑效呢？为什么这一小行命令是必须的呢？

通过上面的映射你可以用三个按键直接打开~/.vimrc文件。一旦你的手指输了掌握了这个命令，你打开这个文件会非常的迅速。

当你在编程的过程中发现了一个新的可以节省时间的快捷键，那么现在就可以很方便的把它加入到你的vimrc文件里了。

元效率

==========

更方便的让自己变得高效才是高效的根本所在。

考虑这样一种方式：假设你现在想成为一个很好的电子摄影师。当你在联系拍照，并且想看看照片的效果时，你会

- 拍照
- 上传到你的电脑上
- 打开图片查看效果
- 这个过程需要好几分钟，如果你可以提高效率呢？

假设你花了50美元买了一个Eye-Fi内存卡。Eye-Fi内存卡是一种自带wifi网卡的内存卡，当你拍一张照片的时候，它会立马上传到你的电脑里。

显然，这是一种幻想，但是这不是很棒吗？

你花了50美元加上一个小时的代价，不过现在你只需要几秒钟就可以坚持你的照片，而非是好几分钟。

假设你的自由职业工作每小时可以获得100美元，你需要工作大概一个班小时来让这个投资得到回报。加入每张图片省下50秒钟的时间，那么你需要照109张照片来支付你的投资。

109张照片并不是什么，它们只不过是九牛一毛。一个甚至不会出现在雷达上的光点。它只不过是你一天工作的很少一部分，你甚至都不会注意到它。现在你两个小时的时间可以拍下远远超过109张照片。

这个问题同样适合于我们的新映射的建立。每次我们使用它的时候都能省下几秒钟的时间，但是建立它的时间远远小于一个小时。

source mapping

==========

一旦你向~/.vimrc文件里添加了映射时，它并不会立马生效。你的vim文件之后再vim启动的时间加载的。也就是说你在添加了映射后需要手动重启一下vim来生效，这个很蛋疼。

我们来用一个映射搞定它： `:nnoremap <leader>sv :source $MYVIMRC<cr>`

这个命令命名为：“加载vimrc文件”

source告诉vim读取指定的文件，并把它当做vim脚本执行。

现在我们可以在疯狂编写代码的同时添加映射了：

- 用ev来打开vimrc文件
- 添加映射
- 用ZZ命令写入文件并退出新窗口，会到原来的窗口
- 用source命令来加载vimrc文件，让我们的改动生效

现在仅需8个按键和编辑映射的操作就可以完成新的映射，并且减少对当前工作的打断。

练习

==========

- 把编辑vimrc文件和加载vimrc文件的映射加入到你的vimrc文件里
- 试试上面的映射，每次添加一些虚假映射
- 阅读:help myvimrc

缩写

==========

Vim有个缩写词的功能，它类似于按键映射，但是只能用于插入，替换和命令模式。它不仅功能强大而且极具扩展性，不过我们这里只讲解最常用的功能。

我们这本书里只讨论插入模式下的缩写，运行下面的命令：

```
:iabbrev adn and
```

现在进入insert模式，然后输入：

```
One adn two
```

在你输入完adn后面的空格之后，vim就换将adn替换为and

缩写的最主要的用处是用来进行输入校正。运行下面的命令:

```
:iabbrev waht what
```

```
:iabbrev tehn then
```

再次进入insert模式，输入下面的文本：

Well, I don't know waht we should do tehn.

现在两个缩写都起作用了，哪怕你在第二个缩写后面没有输入空格。

关键词

==========

如果你当前输入的内容不是关键词的话，vim会用缩写来代替你的输入。非关键词就是那些不在iskeyword选项里的词。运行下面的命令：`:set iskeyword?`

你会看到类似于iskeyword=@,48-75,_,192-255的输出。这个格式比较复杂，但是它大致上说明了以下所有的词都是关键词:

- 下划线；
- 所有的ACSII字母，包括大小写，包括它们重读的版本；
- 所有的在48到57之间的ASCII字符（即是0-9）；
- 所有的在192到255之间的ASCII字符（一些特殊的ASCII字符）。

  如果你想阅读关于这个选项的具体描述，你可以用:help isfname命令，但是我提醒你如果你不想读的比较痛苦的话，最好在阅读之前喝点酒。

  如果只是想用缩写的话，你只要记住只要你输入的不是字母，数字或者是下划线的话，缩写都会有效的。

缩写的一些应用

==========

缩写不仅仅只是用来进行输入校正，它还有其他的功能。让我们添加一些可以在实际的文本编辑里用到的东西。运行下面的命令：`:iabbrev @@ steve@stevelosh.com` `:iabbrev ccopy Copyright 2011 Steve Losh, all`

把上面的邮箱和名称换成你自己的，然后在insert模式下试试。

这个缩写可以用几个简单的字符代替你经常要输入的一大段文本，这个会节省你很多时间的，并且会减少对你的手指的磨损。

为什么不用映射呢？

==========

如果你认为缩写和映射差不多的话，这是对的。但是它们的用途是不一样的。我们看看下面的例子：`:inoremap ssig --<cr>Steve Losh<cr>steve@stevelosh.com`

这个映射可以让你很快的输入自己的签名，在插入模式下试试。

这个命令也确实很不错，但是存在一个问题。在插入模式下输入下面的文本：

```
Larry Lessig wrote the book "Remix".
```

你会发现Vim会扩展Larry的名字里的“ssig”。映射是不会考虑映射字符前面或者后面的字符的——它只对你输入的字符串进行映射。

删除上面的映射，用下面缩写来代替上面的功能: `:iunmap ssig` `:iabbrev ssig --<cr>Steve Losh<cr>steve@stevelosh.com` `再试试缩写`

现在vim会考虑你输入的缩写词的前后的文本，并且只会在需要的时候进行替换。

练习

==========

- 在你的vimrc文件里加入你经常输错的文本校正缩写。并且确保用上一节的映射功能来编辑你的vimrc文件
- 添加你的个人邮箱，网站和签名的缩写词。
- 罗列出你经常输入的文本，为它们建立缩写词。

高级映射

=======

到现在为止，我们已经学了不少关于映射的知识了，我们现在来讨论一下该怎么使用它们。映射功能可以大大提高我们进行文本编辑的效率，所以多花点时间来讨论一下是很值得的。

一个我们已经在前面的例子里介绍的概念是多字符映射，下面我们来看看这个功能。 运行下面的命令： `:nnoremap jk dd`

现在先进入normal模式，然后快速输入jk，vim将会删除当前行。

现在只输入j，并暂停一小会。如果你不是在输入j后立马输入k的话，vim就认为你不想使用映射，所以就只运行了j的基本功能（往下移动一行）。

不过这个映射会让我们在文本里移动的时候显得比较麻烦，所以要用下面的命令删除它： `:nunmap jk`

现在你再次输入jk时会上下移动光标。

一个更加复杂的映射

=======

现在我们已经讨论了很多不同的映射了，是时候让我们一起看看一些比较复杂的映射了。运行下面的命令： `:nnoremap <leader>" viw<esc>a"<esc>hbi"<esc>lel`

这个映射比较复杂！首先，让我们先试试它。进入normal模式，把光标放在文本里的一个单词上，然后输入leader键和引号。Vim会在这个单词的两边加上引号！

这个是怎么起作用的呢？让我们把上面的命令拆开来分析一下：

`viw<esc>"a<esc>hbi"<esc>lel`

- vim 在visual模式下选中当前单词
- esc 退出visual模式，使光标停留在单词的最后一个字符上
- a 进入插入模式，让光标移动到当前字符的后面
- " 我们现在在insert模式下，所以会输入引号
- 进入到normal模式
- h 向左移动一个字符
- b 移动到当前单词的开头
- i 进入到insert模式，并且光标在当前字符前面
- " 在文本里插入引号
- 返回到normal模式
- i 向左移动一个字符，并且光标停留在单词的首字母上
- e 移动到单词的结尾处
- l 向右移动一个字符，使得光标停留在第二个引号后面

注意：因为我们用的是nnoremap而不是nmap，所以不管我们使用了什么字符来进行映射，vim都会使用该字符的最基本的功能。

现在你应该看到了vim映射的潜在的强大了吧，不过你也应该知道它的可读性是多么的差。

练习

=======

- 创建一个和上面差不多的映射，不过用单引号代替双引号
- 用vnoremap来建立这样一个映射：它可以折叠你在visual模式下选择的任意在引号里的文本。你可能需要使用gv命令，所以先阅读一下:help gv。

    - 把H映射为把光标移动到当前行的开始处，因为h的功能是向左移动一个字符，你可以考虑H为h功能的加强版
    - 把L映射为把光标移动到当前行的结尾处，因为l的功能是向右移动一个字符，你可以考虑L是l功能的加强版
- 用:help H和help:L来查看你刚刚覆盖的两个功能，看看你是否需要它们（很有可能你不需要）
- 把上面的映射添加到你的vimrc文件里，并确保你使用的是“edit ~/.vimrc”和“source ~/.vimrc”映射。

训练你的手指

=======

从这章开始，我们开始讨论怎样有效地学习vim，不过我们首先得做一些准备工作。

我们先建立一个很有用的映射，它会大大减少对你左手指的磨损，这个是其他任何映射都是没法做到的。运行下面的命令：`:inoremap jk <esc>`

现在进入insert模式，并输入jk。vim就会认为你输入了escape键，并切换到normal模式

在vim里有多种方式可以退出inset模式：

- escape键
- Ctrl+c
- ctrl+[

但是这几个命令都让你的手指操作起来很不舒服。用jk就很棒了，因为这两个键正处在你最有力的两个手指下面，而且这连个按键很近你不需要把手指张的很开。

有些人喜欢用jj而不是jk，但是我更喜欢jk，主要有两个原因：

- 首先jk用两个按键，这样比用一个按键更有节奏
- 如果在没有映射的环境下，按下jk只会往下移动一行然后再往上移动一行光标，这样你还是在原来的地方，如果用jj就会跑到文本的其他地方去。

学习这个映射

=======

现在有了个很有用的映射，但是我们怎么来记住它呢？不过更有可能的是你的手指已经记住了这个映射，所以当你在集中精神编程的时候，你会毫不犹豫的使用它的。

一个更好的学习这个映射的方式就是来屏蔽escape原有的功能，来强迫你记住新的映射。运行下面的命令：`:inoremap <esc> <nop>`

这个映射很有效，它现在屏蔽了escape原来的功能，通过把映射到（no operation无操作）。

现在你不得不用jk映射来退出insert模式了。当你忘记这个映射，并且按下escape键，然后输入你要在normal模式下输入的命令时，只会在你的文本里多出一些无用的字符。

这个开始的时候会比较纠结，不过你很快就会发现你的大脑和手指很快就记住了这个映射，并且一两个小时内都不会错误地去用escape键。

这个主意很适合用来记住新的映射，甚至在生活中也是这样的，当你要改变一个习惯的时候，只需要让那个习惯的时候变得很难去做就可以了。

如果你想要自己做饭，而非只是用微波炉来加热冷冻快餐，你只需要在每次购物的时候不去购买冷冻快餐就可以了。当你饥饿的时候，你就会开始自己做饭了。

如果你想要放弃抽烟，那么你就一直把烟放在车子的后备箱里。当你想要抽烟的时候，你就会因为需要走出车子，并到后备箱里去拿烟而蛋疼，所以慢慢的你就不想去抽烟了。

练习

=======

- 如果你发现你在visual模式下还是在用方向键来移动光标的话，那么你就把方向键映射成。

- 如果你发现你在insert模式下还是在用方向键来移动光标的话，那么你也把他们映射成，正确使用vim的方式是只要是只要你能在normal模式下移动文本，就不要停留在insert模式下。

本地缓存选项和映射

=======

现在让我们一起花些时间来回归一下我们学的三个功能：映射、缩写和配置项，但是换一种方式：我们每次都在一个缓存里来设置它们。

缓存真正的强大要到下一个章节你才会知道，不过本章我们先学一些基础的东西。

现在，你需要打开两个文本，每一个文本一个独立的窗口。两个文件分别命名为foo和bar，不过你可以随意命名它们。分别向两个窗口里输入文本。

映射

=======

切换到foo文件里，运行下面的命令： `:nnoremap <leader>d dd` `:nnoremap <buffer> <leader>x dd`

在foo文件里，确保你在normal模式下，然后输入键+d。vim会删除当前行。这个我们早就知道了。

还是在foo文件里，输入键+x，vim又删除了一行文本。这个貌似没什么新奇的，因为我们把+x映射成了dd的功能。

现在切换到bar文件里。在normal模式下，输入键+d。同意的vim会删除一行文本。也没啥稀奇的。

现在来看看不同的地方：还是在bar文件里，输入键+x。

什么都没有出现！

第二个映射命令里的告诉vim这个映射只会在定义它的那个缓存里有效。

本地**Leader**键

=======

在上面的例子里，我们用了+x来设置本地缓存映射，但是这个是个不好的方式。一般来说，当你要创建一个供特定缓存使用的本地映射时，你应该用而不是用。

分别用两个不同的前导键，提供了一个类似于“命名空间”的功能，这样可以让不同的映射不至于混淆。

尤其是当你在给其他人提供插件的时候，用来做本地映射就不会让你的插件覆盖了其他人的花了不少时间记住的键的映射。

设置

=======

在前面的章节里，我们讨论了用如何用set命令来设置配置项的。有些选项需要定义成全局的，但是有些则是设置给对应的缓存的。

切换到foo文件里，运行下面的命令： `:setlocal wrap`

再切换到bar文件里，运行下面的命令： `:setlocal nowrap`

现在把vim的窗口调小，你会发现当文本超过窗口的宽度的时候，foo文件里的文本会自动换行，而bar里的文本则不会。

再试试另外一个选项，切换到foo文件，运行下面的命令： `:setlocal number`

现在，切换到bar文件，运行下面的命令： `:setlocal nonumber`

你会发现foo文件里有行号，而bar文件里没有

覆盖

=======

在我们继续下一节之前，先看看一个本地映射的一个有趣的特点。切换到foo文件里，运行下面的命令： `:nnoremap <buffer> Q x` `:nnoremap Q d`

在foo文件里输入Q，会出现什么呢？

当你输入Q时，Vim会运行第一个命令，但为什么不是第二个呢？这是因为第一个映射相对于第二个而言更加明确。

切换到bar文件里输入Q，你会发现vim用了第二个映射，因为在这里它没有被第一个所覆盖。

练习

=======

- 阅读:help setlocal
- 阅读:help map-local

Autocommands

现在我们要讨论一个和映射差不多主要的话题：autocommands

Autocommands是一个告诉vim在发送特定事情的时候运行特定的程序的一个方式。让我们先来看看一个例子。

首先，用:edit foo打开一个新文件然后马上用:quit来关闭它。看看你的磁盘，你会发现这个文件并不存在。这是因为当你新建一个文件的时候如果你没有保存的话，vim是不会真正去创建这个文件的。

下面让我们来改变vim的这个行为，让它在只要你编辑文件的时候就立马创建它。运行下面的命令：

```
:autocmd BufNewFile * :write
```

这个命令涉及的东西比较多，先试试看会有什么功能。再次运行:edit foo，然后用:quit关闭，再看看你的磁盘。这次文件被创建了（不过是空的）。

你需要关闭vim来移除这个autocommand，后面我们会讨论这个在vim里移除的。

Autocommand的结构

下面我们来看看刚刚创建的autocommand的结构：

```
    :autocmd BufNewFile * :write
                        ^          ^ ^
                        |           |  |
                        |           |  The command to run.
                        |           |
                        |          A "pattern" to filter the event.
                        |
                  The "event" to watch for.
```

autocommand的第一部分是我们要关注的事件类型。vim提供了多种可以关注的事件。它们包含：

- 开始编辑一个新的文件
- 读一个文件，无论这个文件是否存在
- 切换一个缓存的文件类型
- 一段事件没有按下任何键
- 进入insert模式
- 退出insert模式

以上只是vim支持的事件的一些很小的例子——vim里有更多的可以用来完成一些有趣的功能的事件。

接下来的部分是用来给执行的autocommand指定更明确的信息的一个模式 。打开一个新的vim实例，运行下面的命令：

```
:autocommand BufNewFile *.txt :write
```

这个命令和上面的功能差不多，但是这次只会对后缀为.txt的文件有作用。

试着用:edit bar，然后用:quit，再试试用:edit bar.txt，然后用:quit。你会发现vim创建了bar.txt文件，但是没有创建bar文件，因为bar不匹配前面的模式。

命令的最后一部分是当事件发生时我们要运行的命令，这个就很明显了，除了一点要注意的，在这里不能使用类似于之类的特殊字符串。关于这个限制我们后面会讨论的，但是现在你必须记住它。

另外一个例子

现在我们来定义另一个autocommand，这次我们用一个不同的事件。运行下面的命令：`:autocmd BufWrite *.html :normal gg=G`

这里面用的命令有些超前，在后面的章节里我们才会讨论normal命令，不过这里你只能将就我了，因为这是个很有用的例子。

打开一个新的文件，命名为foo.html。并且输入下面的文本，要保证一模一样，包含空格：

```
Hello!
```

现在用:w来保存文件，发生了什么呢？vim在关闭文件之前自动进行了格式化。

现在你只需要相信我:normal gg=G命令会格式化当前的文件。暂时不用担心它是怎么起作用的。

我们需要关注的是这个autocommand。它的事件类型是BufWrite，它表示当有文件要写入的时候发生的实际。

我们用*.html这个模式来保证这个命令只会对html后缀的文件起作用。这个让我们的autocommand只对特定的文件执行，这是个很强大的功能，我们后面会继续探讨的。

3、多事件监听

你可以在一个命令里监听多个事件，事件之间用逗号分隔。运行下面的命令: `:autocmd BufWrite,BufRead *.html :normal gg=G`

这个命令和上面的命令差不多，只不过它对文件打开和关闭的操作都会执行格式化的动作。这个命令在当我们和那些不格式化HTML的人一起工作时非常的有用。

在vim脚本里比较常用的一种方式就是，使用BufRead和BufNewFile事件对来操作某种类型的文件，无论是这个文件已经存在还是不存在。运行下面的命令： `:autocmd BufRead,BufNewFile *.html :setlocal nowrap`

这个命令的功能是：当你在打开HTML文件进行编辑的时候，它就会关闭自动换行的功能。

文件类型事件

文件类型事件是比较常用的事件之一。这个事件发生在Vim设置一个文件的filetype时。

我们来对不同的文件类型建立一些有用的映射。运行下面的命令？ `:autocmd FileType javascript nnoremap <buffer> <localleader>c I//` `:autocmd FileType python nnoremap <buffer> <localleader>c I#`

打开一个javascript文件（以.js结尾的文件），挑选一行，然后按键+c，这个命令会注释当前行。

现在打开一个python文件（以py结尾的文件），挑选一行，然后按c，这个命令也会注释当前行，但是这次用的是python的注释符号。

用autocommand和之前学到的本地缓存映射组合起来，可以针对我们所编辑的不同类型的文件创建不同的映射。

这个减少了我们在编码时需要考虑不同类型的文件的动作。当需要注释时我们只需要考虑“注释这一行”而不需要把光标移动到当前行的开始处，然后插入一个注释符。

你可能会发现，执行完上面的命令后，我们会停留在insert模式。不行的是我们现在无法修复这个功能，后面我们会讨论这个问题的。

练习

- 简要阅读一下:help autocmd-events看看有哪些事件可以监听。你不需要记住没一个事件，只是了解一下即可。

- 用setlocal对你常用的几种文件类型创建一些有用的autocommand。一些你需要对不同文件类型自定义的选项是：warp，list和numer。
- 对你常用的文件创建一个快速注释的命令
- 把上面的autocommand加入你的vimrc文件里，当然要用之前的快捷键，要快！

缓存的本地缩写

上一章的内容比较纠结，这章我们看一些比较简单的东西。我们已经学习了怎么定义本地缓存的映射和配置项，现在让我们看看缓存的本地映射。

再次打开foo和bar文件，切换到boo文件，并且运行下面的命令： `:iabbrev <buffer> --- &mdash;`

在foo文件里，进入insert模式，输入以下文本：

```
Hello --- world.
```

Vim会帮你把---替换掉。现在切换到bar文件，vim没有进行替换，这个你现在应该不持久了，因为我们是在foo文件的缓存里定义了本地缩写词。

autocommand和缩写词

我们来把这些本地缓存的缩写词和autocommand配对起来，来组成一个小的系统。

运行下面的命令： `:autocmd FileType javascript :iabbrev <buffer> iff if ( ) {}<left><left><left><left><left>` `:autocmd FileType python :iabbrev <buffer> iff if:<left>`

打开一个Javascript文件然后输入缩写词iff。然后打开一个Python文件，也试试这个缩写词。Vim会根据不同的当前文件的类型进行不同的缩写词替换。

练习

- 创建一些新的缩写片段词用来替代你在某些特定文件里经常做的操作。一些可以替代的缩写有大部分语言里的return，javascript里的function，已经HTML文件里的“和”
- 记住最好的这些技巧的方式是禁止它们原来的操作方式。运行:iabbrev return NOPENOPENOPE会强行让你用缩写来代替原来的操作。添加这些训练的代码到你的vimrc脚本里来节省时间。

自动命令组

在前面的章节里，我们学习了自动命令。运行下面的命令： `:autocmd BufWrite * :echom "Writing buffer!"`

现在试着将缓存的内容写入文件，然后运行:messages来查看消息日志。你将会看到"Writing buffer!"的日志。

再次将缓存的内容写入文件，然后运行:message，你会看到“Writing buffer!”出现了两次。

现在继续运行相同的命令：

```
:autocmd BufWrite * :echom "Writing buffer!"
```

再次将当前缓存的内容写入文件，然后运行:message。你将会看

到“Writing buffer!”的日志出现了4遍。到底是什么原因导致的呢？

当你创建像上面那样的命令时（即是和之前命令一样的命令），vim无法区别你是要覆盖之前的命令，还是要新建一个命令，所以只好把它们当做两个不同的命令来执行。

问题

你现在知道了，在vim里是可以重复定义自动命令，你也许回想：“怎么搞？就这样么！"

真正的问题在于当你每次重新加载你的vimrc文件的时候，里面定义的自动命令也会重复加载，这样的话就会导致vim执行大量的重复命令，从而导致vim的运行速度很慢。

为了模拟上面的问题，运行下面的命令： `:autocmd BufWrite * :sleep 200m`

现在试着写入文件，或许这一点点延时你还感觉不到。现在运行下面的命令3次以上： `:autocmd BufWrite * :sleep 200m` `:autocmd BufWrite * :sleep 200m` `:autocmd BufWrite * :sleep 200m`

再次试着写入文件，现在的延时就很明显了吧。

不过很显然你不会在自动命令里不做其他的事，而只是休眠一会，但是一个vimrc文件很有可能超过一千行，并且很多都是自动命令。这些自动命令和你安装的插件里的自动命令加起来，肯定会降低你的vim的性能的。

自动命令组

不过对于上面的问题，vim提供了解决办法。第一步是把相似的自动命令放到同一个组里面。 打开一个新的vim实例，这样可以清空之前的自动命令，然后运行下面的命令： `:augroup testgroup` `: autocmd BufWrite * :echom "Foo"` `: autocmd BufWrite * :echom "BAR"` `:augroup END`

中间两行的空格没什么意义，只是用来排版的，如果你不喜欢的话也可以不输入。

把一个缓冲写入文件，在用:messages查看一下消息列表。你会看见“Foo”和“Bar”。现在再运行下面的命令： `:augroup testgroup` `: autocmd BufWrite * :echom "Baz"` `:augroup END`

现在猜猜如果你再次把缓冲的内容写入文件，将会有几条消息呢。你会猜是一条，把缓存里的内容写入然后运行：messges来看看你猜的是否正确。

清除自动命令组

当你写入文件的时候发生了什么呢？而你期望的又是什么呢？

如果你认为vim会自动用后面的自动命令组来替换前面的呢，那你就猜错了。不过别担心，大部分人第一次都这么认为。

当你多次用augroup来定义同一个自动命令组时，vim会把这些组的内容。

如果你想清除一个命令组，那么你可以在命令组的定义里用!autocmd来清除她。运行下面的命令：`:augroup testgroup : !autocmd : autocmd BufWrite * :echom "Cats" :augroup END`

在试着将缓存里的内容写入，这时vim只会输出“Cats”。

在**vimrc**文件里使用自动命令

现在我们知道了怎么来组合自动命令已经清除这些组合，我们可以用它来往我们的vimrc文件里添加自动命令，并且保证不会再每次载入vimrc文件的时候都添加重复的命令。

添加下面的内容到你的~/.vimrc文件里: `augroup filetype_html autocmd! autocmd FileTpe html nnoremap <buffer> <localleader>f Vatzf augroup END`

首先开始filetype_html，先清除它，然后定义命令，最后结束定义。如果我们再重新加载vimrc文件，这个清除的动作会防止Vim添加重复的命令。

练习

- 检查一下你的vimrc文件，保证每一个自动命令都是在一个分组里。你可以把多个你命令放在同一个分组里，如果这个分组对你而言是有意义的话。
- 试着找出最后一个映射的功能。
- 阅读 :help autocmd-groups。

1、操作符-区间 映射

这个章节我们继续来探索vim映射的神奇功能：“操作符-区间 映射”，让我们先来弄懂这个名词的意义，然后在使用他。

一个操作符是一个等待你来输入一个移动命令的命令，然后对你当前位置到移动后的位置之间的文本做操作。

常见的操作符有d，y，c。例如：

```
          Operator
          vvvvvv
dw    " Delete    to next word
ci(     " Change  inside parents
yt,     " Yank      until comman
                    ^^^^^^^^
                     Movement
```

在vim里，你创建的移动命令可以和所有现存的命令配合使用。运行下面的命令：`:onoremap p i(`

现在把下面的文本输入到vim里：

```
return person.get_pets(type="cat",fluffy_only=True)
```

把光标放在“cat”上，然后输入dp，会发生什么呢？vim会删除括号里的文本。你可以认为这里面的动作是“选中参数”。

上面的 onoremap 命令让vim是告诉vim当他在等待一个动作时，如果输入了p，那么就把它映射成i(。当我们输入dp的时候，就像是在说“删除参数”一样，对vim而言就是“删除括号里的内容”。

我们可以对所有的命令使用这种新映射方式，把之前的文本再次输入vim：`return person.get_pets(type="cat", fluffy_only=True)`

把光标放在“cat”上，然后输入cp，这次又发生了什么呢？Vim还是会删除括号里的文本，不同的是这次vim会停留在inset模式，因为你用了“替换”命令，而不是“删除”命令。

下面我们看看另外一个例子，运行下面的命令：`:onoremap b /return<cr>`

现在在vim里输入以下文本:

```
      def count(i):
           i += 1
           print i

           return foo
```

把光标移动到第二行的i上面，然后按下db。会出现什么呢？Vim会删除函数体里的内容，一直到“return”语句，这是因为我们的映射使用了vim的normal模式下的搜索。

当你在考虑怎么来定义一个新的操作符-区间的动作时，你可以按照下面的思路来考虑：

- 从当前位置开始；
- 进入visual模式；
- 然后是动作的映射
- 最后你要操作的文本都已经选定了

你要做的是把第三步替换成你想要的功能。

改变初始位置

对于我们之前学习的功能，你可能发现了一个问题，如果我们的操作只能从当前位置开始的话，这样会限制我们想要做的。

不过Vim并不会限制我们能过什么，所以这个问题还是有办法来解决的。运行下面的命令：`:onoremap in( :<c-u>normal! f(vi(<cr`

这个命令看起来很难，不过我们可以先试试这个命令。把下面的文本输入到vim里：`print foo(bar)`

把你的光标移动到单词“print”的任何位置上，然后输入cin(。Vim会删除括号里的内容，然后把你的光标放在中间，并且是处于insert模式下。

你可以把这个映射当做“在下一个括号里”，它会对当当前行的下一个括号里的内容执行操作符所对应的操作。

接下来我们来做一个和上面相对的命令“在上一个括号里面”。运行下面的命令：`:onoremap il( :<c-u>normal! F)vi(<cr>`

你自己可以输入一些文本来测试上面的映射，确保它确实有效。

那么这个映射到底是怎么工作的呢？首先，是一个现在可以暂时不考虑的命令——只要知道它的功能是为了让这个映射在所有的情况下都能正常工作的。剔除的话，就剩下：`:normal! F)vi(<cr>`

:normal!使我们后面将会讨论的一个命令，现在只需要知道它的功能是模拟在normal模式下按下键盘。例如:normal! dddd会删除两行文本，就像按下dddd一样。末尾的表示执行输入的命令。

所以排除上面的那些命令，就只剩下 `F)vi(`

现在就变得很简单了：

- F) :移动到最近的')'字符；
- vi(:进入visual模式，并且选中括号里的文本。

最后我们在visual模式下选中了我们要操作的文本，vim接着就会执行我们指定的操作。

常用的规则

操作符-区间 映射的创建方式多种多样，记住下面两条准则可以帮助你方便的来创建操作符-区间映射：

- 如果你的映射最后的结果是在visual模式下选择了一些文本的话，那么vim会对这些文本进行操作；
- 否则的话，vim会对光标之前的位置和当前位置之间的文本进行操作。

练习

- 为“靠近下一个括号”和“靠近上一个括号”建立操作符-区间映射；
- 对大括号建立上面四种映射；
- 创建一个“在下一个邮件地址里”的映射，这样的话你就可以来修改下一个邮件地址。in@是一个对实现这个映射有帮助的命令。不过你可能会用/....正则表达式..来实现。
- 阅读:help omap-info 看看你能否找到的作用。

状态栏

Vim允许你自定义文本区下面的状态栏里的文本，这个功能是通过设置statusline选项来实现的。运行下面的命令： `:set statusline=%f`

你会看见当前文件的路径（相对于当前文件夹的路径）在状态栏里。现在运行下面的命令： `:set statusline=%f\ -\ FileType:\ %y`

这次你会在状态栏里看到类似于“foo.markdown - FileType:[markdown]”的内容。

如果你对C语言的printf函数或者Python的字符串插值比较了解的话，这个选项就会比较好理解了。如果不熟悉的话，只要记住%开头的文本都是会根据后面的字符进行替换的。在我们的例子里%f被替换为当前的文件名，%y被替换为当前的文件类型。

注意用\转义的空格，因为set命令可以一次设置多个选项，选项间是用空格分开的，所以这里的空格是需要转义的。

状态栏很容易就会变得非常复杂的，所以用一个好的方式来设置的话，我们也会比较清楚的。运行下面的命令：

```
:set statusline=%f      "当前文件的路径
:set statusline+=\ -\    "分隔符
:set statusline+=FileType:"标签
:set statusline+=%y     "当前文件的类型
```

第一行我们用=号来清除statusline之前存在的设置。在后面的命令里我们用+=来依次拼接我们的选项。我们同时会对每一行的设置都加上注释，方便别人和自己阅读。

运行下面的命令:

```
:set statusline=%l      "当前行
:set statusline+=/     "分隔符
:set statusline+=%L     "总行数
```

现在状态栏只包含文本的当前行号和总行好，看起来就向“12/223”一样。

宽度和填充

一些其他的字符可以用在%后面来改变信息的显示方式，运行下面的命令： `:set statusline=%4l`

现在行号的显示宽度是4，在行号不足4位的时候前面会自动填充空格的，例如“ 12”，这样的话，就可以防止状态栏的文本显得比较混乱了。

默认情况下，填充的空格会加在值的左边。运行下面的命令： `:set statusline=Current:\ %4l\ Total:\ %dL`

现在，你的状态栏可以看起来和下面一样：

```
Current:    12 Total:  223
```

你可以用-来让填充的空格出现在右边，运行下面的命令: `:set statusline=Current:\ %-4l Total:\ %-4L`

现在你的状态栏应该看起来和下面的差不多：

```
Current: 12  Total: 223
```

现在看起来好多了，因为数字和标签考的比较近。

对于数值可以显示告诉Vim用0来作为填充符，运行下面的命令： `:set statusline=%04l`

现在当你的光标在第12行的时候，你会在状态栏里看到“0012”。

最后，你可以限制输出值的长度。运行下面的命令： `:set statusline=%F`

%F会显示当前文件的全路径，选择用下面的命令来改变显示的最大宽度： `:set statusline=%.20F`

如果显示的路径长度超过做大长度的话，就会被截断的，就像下面一样： `<hapters/17.markdown`

这个功能很有用，它可以防止某些字符输出太长而占用整个状态栏。

一般格式

状态栏的一般格式在:help statusline定义了：

`%-0{minwid}.{maxwid}{item}`

除了%外，其他的项都是可选的。

分开设置

后面我们会继续讨论状态栏的设置的，不过我们还需要讨论一种简单但是很有用的设置。运行下面的命令：

```
:set statusline=%f          "当前文件的路径
:set statusline+=%=        "切换到右边
:set statusline+=%l          "当前行号
:set statusline+=/            "分隔符
:set statusline+=%l         "总行号
```

现在状态栏里既包含了当前文件的路径，也包含了当前行和总行数，分别在左边和右边显示。%=告诉vim所有在这个后面设置的选项都会显示在状态栏的右边。

练习

- 浏览:help statusline里的选项，不用担心有些你不知道的东西。
- 在你的vimrc文件里定义一个你自己的状态栏。一定要使用+=来分隔多行设置项，对于每个设置项要加上注释。
- 试着用autocmd和setlocal命令来对不同的文件类型设置不同的状态栏，记得要对autocmd进行分组，以免重复。

响应式编程

现在为止我们已经讨论了很多可以用来提高vim效率的命令。除了自动命令分组意外，其他的命令都是单行的，并且是可以直接添加到vimrc文件里的。

在这本书的后面内容里，我们将会从一门真正的编程语言的角度来讲解vim脚本，但是在开始编写vim脚本之前，我们先来讨论一下怎么在写大段代码的时候保持思路清晰。

注释

vim脚本的功能非常强大，但是经过这么多年，已经发展的成一个很容易让人迷乱的语言。

选项和命令往往都很简洁，也导致它们的可读性变得很差，同时为了保持兼容性，有大大增加了代码的复杂度。写一个用来帮助用户自定义vim的插件有带来了另一层的复杂性。

当你写的vim脚本行数比较多的时候就得要细心了，最好能加个注释表明具体的功能，如果有相关的材料最好说出来。

这不仅仅是为了能够让你在几个月甚至是几年后能够来维护它，如果你把在GitHub上分享你的vimrc文件的时候，它也能够帮助别人理解你的代码。

分组

我们对于编辑和加载vimrc文件的映射能够让我们很快的来往vimrc文件里添加新的配置。但是这样也会导致配置项比较杂乱，并且无法控制和管理。

一个比较好的方式来防止上的问题就是用vim的代码折叠功能把代码行分成不同的组。如果你之前没有用过vim的代码折叠功能，现在看看下面这个很不错的的教程。

首先我们要设置vim的代码折叠功能。把下面的脚本加入到你的vimrc文件里：

```
augroup filetype_vim
  au!
  au FileType vim setlocal foldmethod=marker
augroup END
```

这个会告诉vim对所有的vim脚本文件用marker方法来折叠代码。在你的vimrc文件的窗口里运行:setlocal foldmethod=marker。直接重新加载vimrc文件是没有作用的，因为vim已经设置了这个文件的类型，所以文件类型的事件是不会触发的，因此自动命令也就不会执行了。不过以后你就不需要手动来设置这个选项了。

现在在自动命令的前面和后面加上一些文本，使得它看起来和下面差不多：

```
" Vimscript file settings ---------------------- {{{
augroup filetype_vim
    au!
    au FileType vim setlocal foldmethod=marker
augroup END
" }}}
```

返回到normal模式下，把你的光标放在任意一行，然后输入za。Vim会把从包含“{{{”和“}}}”的文本行进行折叠。再次输入za会打开代码行。

开始你可能会认为显示地加注释来表示代码折叠显得比较丑陋。我开始也是这么认为的。对于大多数文件我都认为是不对的。因为并不是每个人都用相同的编辑器，所以对于那些用非Vim编辑器来查看代码的人而言这就是增加了干扰。

vimrc文件是个特例，因为很少有不用vim的人会来读你的vimrc文件，尤其重要的一点是，当你在编写vim脚本时，你要显式

地进行分组。

试着这样做一段时间，慢慢加深这个想法。

缩写名称

Vim允许你用大部分命令和选项的缩写词。例如，下面的两个命令的功能是一样的：

```
:setlocal wrap
:setl wrap
```

不过我不建议你在你的vimrc文件和插件的代码里用他们，因为vim脚本本身就足够简洁，而且还比较晦涩——运用缩写词只会让它变得更加难以阅读。即是你知道一个命令的缩写的意义，但是别人也会许不知道。

不过说回来，对于在编码时手动输入的命令，缩写词是很有用的，因为它们可以提高输入效率。因为一旦你输入了换行符，别人是不会再看到的，所以能不多按键盘就尽量不多按。

练习

- 浏览你的整个vimrc文件，把命令行按照 相关性进行分组。例如“基本设置”，“特殊文件设置”，“映射”已经状态栏。为每个区域添加折叠标志。
- 试着找出怎么才能够在每次打开文件时让vim自动对文件文件进行折叠动作。你可以先看看:help foldlevelstart。
- 浏览你的vimrc文件，把所有的缩写词都改成命令的全名。
- 浏览你的vimrc文件，确保没有任何隐私信息，然后把创建一个git或者Mercurial仓库，把vimrc文件放进去，然后添加一个到这个文件的链接。
- 把你刚刚创建的仓库提交到BitBuck或者GitHub上去，让别人能够看见它并且能够参考它。记得要频繁提交当你修改的时候，这样可以记录你修改的地方。
- 如果你在多个机器上用到vim，可以把仓库里的vimrc文件下载下来，然后建立一个符号链接。这样就可以很方便地让你的vim用同一份配置文件了。

变量

到了这里我们已经讲解完了单行命令。在接下来的本书的第三部分里，我们我们将会把vim脚本当做一门编程语言来讲解。这个部分没有之前我们学的那些那么让人兴奋，不过这个会给后面要做的东西打下基础，它贯穿从零创建一个vim插件的整个过程。

首先我们要讨论的是变量。运行下面的命令：

```
:let foo = "bar"
:echo foo
```

vim会输出“bar”。foo现在是一个变量，我们把它赋值为字符串“bar”。现在运行下面的命令：

```
:let foo = 42
:echo foo
```

vim会输出“42”，因为我们给vim赋值为整数“42”。从这里看来vim应该是动态类型的。不过事实上并不是这样，我们和面会讨论这个的。

选项作为变量

你可以像对待一个变量那样来设置和读取一个选项，通过一个特殊的语法。运行下面的命令：

```
:set textwidth=80
:echo &textwidth
```

Vim会输出“80” 。在一个名称的前面加上&表示你要引用一个选项的值，而不是一个恰好有相同名称的变量。

我们再看看vim是怎么处理布尔值的选项的。运行下面的命令：

```
:set nowrap
:echo &wrap
```

Vim会输出“0”，现在试试下面的命令：

```
:set wrap
:echo &wrap
```

这次vim会输出“1”。很显然，vim把整数“0”当做“false”，整数“1”当做“true”。可以认为vim把任意的非零整数都当做“true”来处理， 并且事实上也是这样的。

我们也可以像设置变量那样来设置选项，运行下面的命令： `:let &textwidth = 100` `:set textwidth?`

Vim会输出"textwidth=100"

可以直接用set命令来设置选项，为什么还要用let呢？运行下面的命令： `:let &textwidth = &textwidth + 10` `:set textwidth?`

这次Vim会输出“textwidth=110” 。当你用set来设置选项的时候，你只能把它设置成一个单一的值，但是如果你用let的话，你

可以用vim脚本来任意设定选项的值。

本地选项

如果你想设置本地选项的值，而不是一个全局的，你需要在变量名的前面加上&l:。

在两个窗口里分别打开文件，运行下面的命令： `:let &l:number = 1`

切换到另外一个文件，运行下面的命令： `:let &l:number = 0`

你会看到第一个窗口里有行号显示，而第二个窗口则没有。

本地寄存器变量

你同样可以把寄存器当做变量来进行读写。运行下面的命令： :let @a = "hello!"

现在把光标放在文本的任意位置上，输入"ap。这个命令告诉vim“在当前位置粘贴寄存器a里的文本”。我们刚刚设置了那个寄存器的文本，因此vim会在你的文本里粘贴上“hello!”。

寄存器也可以被读出来。运行下面的命令： `:echo @a`

vim会输出"hello!"

在你的文件里选择一个单词，然后用y命令来复制它，然后运行下面的命令： `:echo @"`

Vim会输出你刚刚复制的命令。"寄存器是无名寄存器，这是你没有指定位置进行复制的文本存储的地方。

在你的文件里用/someword进行搜索，然后运行下面的明： `:echo @/`

vim会输出你刚刚输入的搜索表达式。这个寄存器可以让你用程序来读取和改变当前的搜索表达式，这个功能有时候会非常有用的。

练习

- 浏览你的vimrc文件，把部分set和setlocal命令改用let形式来实现。记住布尔值的选项也是需要被设置的。
- 把类似于wrap的布尔值选项的值设置成不是0和1的值，看看不同的数值会有什么效果呢？如果设置成字符串又会是什么效果呢？
- 把上面的改动都改回去，因为如果set就能够满足需求的话，尽量不要用let，因为那样会变得比较难读。
- 阅读:help registers，看看有哪些寄存器可以使用。

e### 变量作用域

如果你之前是用python或者ruby之类的动态语言的话，那么你现在对vim脚本的变量会觉得很熟悉。除了大部分和你预料的都相同之外，vim添加了一个不同点：作用域。

在两个窗口里分别打开一个缓冲文件，在其中一个窗口里运行下面的命令：

```
:let b:hello = "world"
:echo b:hello
```

正如你所预料的，vim会输出"world"。现在切换到另外一个窗口，运行下面的命令：

```
:echo b:hello
```

这时vim会抛出一个错误，说是找不到这个变量

当我们在变量名前面加:b时，相当于告诉vim变量hello是属于当前缓冲区的一个局部变量。

Vim变量有很多不同的作用域，但是在学习其他作用域之前我们先要学习更多的vim脚本。现在，只要记住，当变量名前面有一个字符加分号做前缀时，就表示它是变量的作用域。

练习

- 浏览变量作用域的列表，在:help internal-variables里。不用担心，如果你不知道它们的意义的话，只需要看看，并有个映像就可以了。

条件分支

每一个编程语言都有实现条件分支的方式，在vim脚本里是通过if语句来实现的，它是vim里来实现分支的主要方式。vim里不会像Ruby里一样有unless分支，所以你代码里的分支都只能用if来实现。

在我们开始讲解vim的if语句之前，我们要花点时间先看看一些语法的东西，以便我们能够达成共识。

多行语句

有些时候你能不够把vim脚本都写在同一行里，这样的例子我们在自动命令分组里见到过。以下使我们之前写过的一段代码：

```
:augroup testgroup
:     autocmd BufWrite * :echom "Baz"
:augroup END
```

你也可以把这三行命令卸载vim脚本的同一行里，这样也是可以的，不过当你手动输入命令的时候就显得比较冗长了。不过你可以用一个|来分割命令。运行下面的命令：

```
:echom "foo" | echom "bar"
```

vim会把上面的命令看做是两行。用:mesages来看看日志里是否有两行信息。

在本书后面的章节里，如果你想手动输入命令，但是不想输入换行和分号，那么你可以把它们放在同一行里，只需要用|分割就可以了。

基本If语句

现在我们可以开始了，运行下面的命令：

```
:if 1
:     echom "ONE"
:endif
vim会输出“ONE”，因为1代表真值，现在运行下面的命令：
:if 0
:     echom "ZERO"
:endif
```

Vim现在不会输出“ZERO”，因为整数0表示假。现在我们看看如果是字符串，会怎么样呢。运行下面的命令:

```
:if "something"
:     echom "INDEED"
:endif
```

这个结果可能会出乎你的意料之外。Vim不一定把一个非空的字符串当作真值来处理，所以它不会输出任何东西。

让我们继续往下探究，运行下面的命令：

```
:if "9024"
:     echom "WHAT?!"
:endif
```

这次vim输出了文本，vim究竟是怎么来处理字符串的呢？

试着去理解一下到底是怎么回事，运行下面的命令：

```
:echom "hello" + 10
:echom "10hello" +10
:echom "hello10" + 10
```

第一行输出10，第二行输出20，第三行又输出10

通过上面的命令的输出，我们可以总结出一些关于vim脚本处理字符串的结论：

在必要的时候vim会强制进行类型转换。当执行10+"20foo"的时候，Vim会把“20foo”转换成一个整数（这里是20），然后来和10进行想加

以数字开始的字符串都会转换成开始的数字，其他的都会转换成0.

在所有的转换都结束，并且条件表达式的执行结果为非零整数时，if里面的语句才会执行。

Else 和Elseif

Vim和python一样既支持else子句也支持else if子句。运行下面的命令：

```
:if 0
:     echom "if"
:else if "nope!"
:     echom "elseif"
:else
:     echom  "finally"
:endif
```

最终vim会输出“finally”，因为之前的条件的值都是0，也就是为false。

练习

- 对于vim把字符串转为整数的行为喝杯酒来安慰自己

比较

我们已经学习过了条件语句，如果我们不进行比较的话，if语句就是没用的。当然vim会让我们进行比较运算的，只不过不是很直观而已。

运行下面的命令：

```
:if 10 > 1
:     echom "foo"
:endif
很显然，vim会输出“foo”，现在运行下面的命令：
:if 10 > 2001
:     echom "bar"
:endif
```

这次vim什么都不会输出，因为10不是大于2001。现在所有的代码都是输出我们预料的结果。运行下面的命令：

```
:if 10==11
:     echom "first"
:elseif 10 == 10
:     echom "second"
:endif
```

Vim会输出“second”，这也没什么特别的。现在我们来比较一下字符串，运行下面的命令：

```
:if "foo" == "bar"
:     echom     "one"
:elseif "foo" == "foo"
:     echom      "two"
:endif
```

Vim会输出“two”，也没什么特别的。这是我之前要讨论的内容么？

大小写敏感

运行下面的命令：

```
:set noignorecase
:if "foo" == "FOO"
:     echom "vim is case insensitive"
:else if "foo" == "foo"
:     echom "vim is case sensitive"
:endif
```

vim会执行elseif里的语句，所以很显然，vimscript是大小写敏感的。知道这个也不错，不过也没什么大不了的。现在运行下面的命令：

```
:set ignorecase
 :if "foo" == "FOO"
:     echom "no, it couldn't be"
:else if "foo" == "foo"
:     echom "this must be the one"
:endif
```

什么？打住，对了正如你所见。

==的结果取决于用户自己的设置。

我保证我不是在忽悠你。再次试一试上面的例子，我不是在开玩笑，这也不是我自己编造的。

防御式编程

这意味着什么呢？它意味着当你在给别人编写插件时，==比较的结果是不可信的。一个简单的==是不该出现在你的插件的代码里的。

这个和之前的nmap和nnoremap的观点一样。不要相信用户自己的设置，vim是一个比较老，广泛使用的，并且比较复杂。所以当你在写代码的时候要考虑到用户的任何可能的设置。

那我们该怎么处理这个复杂的场景呢？事实上，vim有额外的两种比较操作符来处理这个。

运行下面的命令：

```
:set noignorecase
:if "foo" ==? "FOO"
:     echom "first"
:elseif "foo" ==? "foo"
:     echom "second"
:endif
```

Vim会输出“first”，因为==?是表示“无论当前用户的设置是什么，比较都是不区分大小”。现在运行下面的命令：

```
:set ignorecase
:if "foo" ==# "FOO"
:     echom "one"
:elseif "foo" ==# "foo"
:     echom "two"
:endif
```

Vim会输出“two”，因为==#是“无论当前用户的设置是怎样，比较都是大小写敏感的”。

这个例子告诉我们在比较的时候都要显式的用==?或者==#来进行比较。用==来比较是部队的，甚至在某些地方会失效。所以为了减少麻烦，多输入一个字符吧。

当你在比较证书的时候，现在是没必要这样做的。虽然这样我还是觉得最好对所有的比较都用大小写敏感的比较符，即使是在不需要的地方，这样总比在需要的时候忘记去用要好多了。

用==#和==?来对整数进行比较也是没有问题的，如果你以后把整数改成字符串也是能够正常工作的。如果你对整数比较喜欢用==号的话，那么你就要记住，当你把整数改成字符串的时候也要把==改成大小写敏感的比较符。

练习

- 试试用:set ignorecase和:set noignorecase来改变设置，然后看看不同的比较符的结果是怎样的。
- 阅读:help ignorecase去看看为什么需要设置这个选项。
- 阅读:help expr4来看看所有的比较操作符。

函数

就像其他的编程语言一样，vimscript也有函数。我们先来看看怎么创建函数，然后再看看它们古怪的地方。

运行下面的命令：`:function meow()`

也许你会认为这个命令会创建一个名叫Meow的函数，不过遗憾的是，事实上不是这样，我们现在已经碰到了vimscript一个古怪的地方。

在vimscript里，如果函数没有指定范围的话，函数名必要以大写字母开始

即使你给函数加上了范围（后面会讨论怎么加），不过你也最好对函数名以大写字母开始。因为大部分的vimscript程序员都会这样做，所以不要打破这个潜规则。

好了，现在让我们来真正定义一个函数，运行下面的命令：

```
:funciont Meow()
:     echom "Meow!"
:endfunction
```

这样就正确的地定义了一个函数，现在让我们来运行这个函数：`:call Meow()`

正如我们所预料的，vim会输出“Meow!”。

现在让我们试着返回一个值，运行下面的命令：

```
:function GetMeow()
:     return "Meow String!"
:endfunction
```

现在通过下面的方式来运行它：`:echom GetMeow()`

Vim会调用上面的函数，并且把返回值传给echom，这样就会输出“Meow String!”。

调用函数

通过上面已经知道我们有两种方式来调用函数。

当你想直接调用一个函数的时候，可以用call命令，运行下面的函数：

```
:call Meow()
:call GetMeow()
```

第一个会输出“Meow!”,但是第二个不会输出任何信息。当你用call来调用函数的时候，返回值就被丢弃了，所以call在函数有副作用的时候用比较好。

第二个调用的方式就是通过表达式。在表达式里，你不需要用call，只需要用函数的名称来调用。运行下面的函数：

```
:echom GetMeow()
```

就像之前一样，这样会调用GetMeow()然后把返回值传给echom。

隐式返回值

运行下面的命令：

```
:echom Meow()
```

这次会输出两行："Meow!"和"o"。第一个结果显然来自于Meow函数内部的echom。第二个向我们显示了当一个vimscript函数没有返回值时，它会默认返回一个0.让我们利用这个特点，并且运行下面的命令：

```
:function TextwidthIsTooWide()
:     if &l:textwidth ># 80
:          return 1
:   endif
:endfunction
```

这个函数里用了我们之前学到的很多概念：

- if语句
- 把选项当作变量
- 把选项本地化
- 大小写敏感的比较

因为我们没有显示返回一个值，所以vim会默认给个返回值0，也就是false。让我们来修改一下代码，运行下面的命令：

```
:setlocal textwidth=100
:if TextwidthIsTooWide()
:     echom  "WARNNING:WIde text!"
:endif
```

这次函数里的if里的语句执行了，并且函数返回1，所以我们手动输入的if也执行了它里面的语句。

练习

- 阅读:help call，忽略任何超出我们现在学习范围的内容。回答一下，你可以给函数传多少个参数呢？这个是不是有点吃惊呢？
- 阅读:help E124的第一段，找出那些字符是可以用在函数名称里的。下划线可以么？斜杠呢？重音符号呢？unicode字符呢？如果看了文档还不清楚的话，你可以直接在vim里试试。
- 阅读 :help return。看看它的缩写形式是什么呢？是不是你所认为的呢？如果不是，那是为什么呢？

函数传参

Vim函数，当然也可以接受参数，运行下面的命令：

```
:function DisplayName(name)
:     echom     "Hello! My name is:"
:     echom     a:name
:endfunction
运行这个函数：
:call DisplayName("Your Name")
```

Vim会输出两行文本：“Hello! My nameis:”和“Your Name”。

注意传给echom命令的参数前面的a:，这个代表一个变量范围，这个我们之前就讨论过的。

现在让我们去掉这个变量范围的前缀，再看看vim怎么反应的。运行下面的命令：

```
:function UnscopedDisplayName(name)
:     echom     "Hello!     My name is:"
:     echom      name
:endfunction
:call UnscopedDisplayName("Your Name")
```

这次Vim会提示找不到变量name。

当你在写一个vimscript函数，并且接受传参时，你都需要在变量名前加上a:，来告诉vim这是一个本地变量。

可变参数

Vimscript函数可以像Javascript和Python一样接变长参数。运行下面的命令：

```
:function Varg(...)
:     echom     a:0
:     echom     a:1
:     echo     a:000
:endfunction

:call Varg("a","b")
```

这个函数向我们展示了几个事情，让我们来一个一个地看看。

函数里的...告诉vim这个函数可以接受任意数目的参数。这个和Python函数里的*args一样。

第一行输出a:0也就是输出“2” 。当你定义一个函数，它接受一连串的参数时，a:0会被设置成参数的数目。在这个例子里，我们传入了两个参数，所以vim会输出“2”.

第二行输出a:1也就是输出“a”。你可以用a:1，a:2等来引用每个你接收到的参数。如果你用a:2，vim会输出“b”。

第三行有点纠结了。当一个函数的参数是可变参数时，a:000会被设置成包含所有参数的列表。我们现在还没有讲过列表，不用担心，我后面会讲解的。你不用给echom传递一个列表，这就是为什么我们第三行用echo来代替echom了。

你也可以混用可变参数和一般参数。运行下面的命令：

```
:function Varg2(foo,...)
:     echom a:foo
```

```
:     echom a:0
:     echom a:1
:     echo    a:000
:endfunction

:call Varg2("a","b","c")
```

你可以看到vim把“a”赋值给a:foo，然后后面的命令都被放到可变参数列表里。

赋值

运行下面的命令：

```
:function Assign(foo)
:     let     a:foo = "Nope"
:     echom a:foo
:endfunction

:call Assign("test"0
```

Vim会抛出一个错误，因为你不能再次给形式参数赋值。现在运行下面的命令：

```
:function AssignGood(foo)
:     let foo_tmp = a:foo
:     let foo_tmp = "Yep"
:     echom foo_tmp
:endfunction

:call AssignGood("test“)
```

这次，函数起作用了，并且vim会输出“Yep”。

练习

- 阅读:help function-argument的前两章。
- 阅读:help local-variables。

数值

现在是时候来看看在vimsrcipt里可以使用的数据类型。现在我们先看看Vim的数值类型。

Vimscript有两种数值类型：整数和浮点数。整数是一个32字节的有符号整数。浮点数很显然是一个定点浮点数。

数据格式

你可以用几种不同的方式来表示数值。运行下面的命令：

```
:echom 100
```

很显然,Vim会输出“100”，现在运行下面的命令：

```
:echom 0xff
```

这次Vim会输出“255”.你可以用在一个数值之前加上0x或者0X来表示十六进制数。运行下面的命令：

```
:echom 010
```

你也可以用0作为前缀来表示一个八进制数。但是对于这个用法要小心点，因为这种方式比较容易犯错。运行下面的命令：

```
:echom 017
:echom 019
```

对于第一个命令Vim会输出“15”，因为“17”在八进制里的值是“15”.对于第二条命令，Vim把它当作十进制数来处理，即使它是以0开始的，但是它不是一个有效的八进制数。

因为vim在这种情况下会默认取错的值，所以我比较推荐的方式是在vim里尽量不要用八进制数。

浮点数

浮点数也可以用多种格式来表示。运行下面的命令：

```
:echo 100.1
```

注意我们在这里用的是echo而不是常用的echom。后面我们会解释为什么要这样做的。Vim会输出“100.1”，毫无意外。你也可以用指数表达式来表示浮点数。运行下面的命令：

```
:echo 5.45e+3
```

这次会输出“5450.0”。你也可以用复数作为指数，运行下面的命令：

```
:echo 15.45e-2
```

这次vim会输出“0.1545”。

指数签名的+和-是可选的，如果指数前面没有符号的话，那么它的值默认就是正数。运行下面的命令：

```
:echo 15.3e9
```

Vim会输出“1.53e10”，这个和之前的值是等效的。不过e前面的数字里的“.”是不能省略的，即使它是一个整数，运行下面的命令，vim会报错的：

```
:echo 5e10
```

强制类型转换

当你在一个算术表达式，比较表达式，或者其他表达式里同时用到整数和浮点数时，Vim会把整数转换成浮点数，最终返回一个浮点数的结果。运行下面的命令：

```
:echo 2 * 2.0
```

Vim会输出“4.0”。

除法

当两个整数相除时，余数会被丢弃掉。运行下面的命令：

```
:echo 3/2
```

Vim会输出“1”。如果你想要vim进行浮点数除法，那么就需要其中一个数是浮点型的，这样的话，vim会把另外一个强制转换成浮点型。运行下面的命令：

```
:echo 3 / 2.0
```

vim会输出“1.5”，这里3会被强制转换成一个浮点数，然后就会进行浮点数除法运行。

练习

- 阅读:help Float。什么情况下浮点数在vimscript里是无效的呢？
- 阅读:help floating-point-precision。当你写一个vim插件来处理浮点数时，精度有什么意义呢？

字符串

接下来我们要看的一种变量类型是字符串。由于在vim里进行的工作都是进行字符串操作，所以对于字符串变量你会用比较多的。

运行下面的命令：

```
:echom "Hello"
```

字符串连接

对于字符串比较常用的一个功能就是把字符串拼接在一起。运行下面的命令：

```
:echom "Hello, "+"world"
```

会出现什么呢？Vim很奇怪地输出了“0”。

问题的关键在于：Vim里的+只是用于数字的。当你把字符串作为+的一个操作数时，Vim会把先把它转换成数值然后再进行加法运算。运行下面的命令：

```
:echom "3 mice" + "2 cats"
```

这次Vim会输出“5”，因为上面的两个字符串分别被转换成数字“2”和“3”。

我这里说所的数字都是整数，vim是不会把字符串转换成浮点数的！运行下面的命令来证明这个结论：

```
:echom 10 + "10.10"
```

Vim会输出“20”，因为当把“10.01”转换成数字的时候，会吧小数点以及小数点后面的字符都忽略掉了。

如果要进行字符串的拼接，那么就要用字符串的拼接命令。运行下面的命令：

```
:echom "Hello, " .  "world"
```

这次Vim会输出“Hello, world”。“.”是vim里的字符串拼接操作符，它可以用来进行字符串的拼接操作。它不会在两个字符串之间加上空格或者其他的字符。

数字也可以转换成字符串。运行下面的命令：

```
:echom 10 . "foo"
```

Vim会输出“10foo”。首先，10会被转化成一个字符串，然后会和后面的字符串进行拼接。不过，当你对浮点数进行操作时，结果就不会一样了。运行下面的命令：

```
```

```
:echom 10.0 . "foo"
```

这次vim会抛出一个错误，说是不能把浮点数当作字符串来使用。Vim允许你在进行加法操作时，把一个字符串当作浮点数来操作，但是在进行字符串拼接时却不允许把浮点数当作字符串来使用。

这个问题的原因在于vimscript和Javascript有点相似：它让你很方便的进行操作，并且对于类型的限制是非常松的，但是这是一个很不好的方式，因为在后面它会让你很烦恼的。

当你在写vimscript的时候，最好能够弄清楚你的每一个变量的类型。如果你需要改变一个变量的类型，最好用一个函数显式地来实现，即使当时不是严格要求的。不要依赖vim的强制转换，因为有时候它会让你失望的。

特殊字符

和其他许多编程语言一样，vim运行你用转义符来表示一些很难用键盘来输入的字符。运行下面的命令：

```
:echom "foo \"bar\""
```

上面字符串里的\"会被一个双引号来代替的，这个也许是你已经预料到的。

转义字符串基本上都会按照你的意愿来显示的。运行下面的命令：

```
:echom "foo\\bar"
```

Vim输出“foo\bar”，和大多数编程语言一样\是\的转义字符串。现在运行下面的命令：（注意这里用的是echo而不是echom）

```
:echo "foo\nbar"
```

这次vim会输出两行文本，“foo”和“bar”，因为\n代表换行符。

现在运行下面的命令：

```
:echom "foo\nbar"
```

Vim会输出类似于“foo^@bar”的文本。当你用echom来输出字符串而不是用echo时，vim会输出所有的字符，而不是它转义后的文本，也就是说它会输出和echo不一样的文笔。^@是vim用来表示换行符的字符。

不转义的字符串

Vim也允许你使用不转义的字符串来避免转义字符串的作用。运行下面的命令:

```
:echom '\n\\'
```

Vim会输出“\n\”。用单引号表示不对字符串进行转移，除了两个单引号会表示一个单引号这个例外。运行下面的命令：

```
:echom 'That''s enough.'
```

Vim会输出“That's enough”，在不转义字符串里，单引号是惟一一个例外的字符。

在本书的后面（当我们深入讨论正则表达式的时候），我们后面会再次讨论不转移字符串的。

布尔值

你可能会考虑在if语句里的字符串的代表什么布尔值。运行下面的命令：

```
:if "foo"
:     echo "yes"
:else
:     echo "no"
:endif
```

Vim会输出“no”。如果你很奇怪为什么会这样，你可能就需要再次阅读前面的条件语句章节，我们在那里讨论了这个问题。

练习

- 阅读:help expr-quote。看看你在vim可以使用哪些转义字符串。试着找出怎么插入一个制表符。
- 找出一种在一个字符串里插入制表符，但是不用转义字符串的方式。参考:help i_CTRL_V。
- 阅读:help literal-string。

字符串操作函数

Vim有很多内置的字符串操作函数。在这个章节里，我们会讲解一下vim里最常用的几个字符串操作函数。

长度

我们要看的第一个个函数是strlen。运行下面的例子：

```
:echom strlen("foo")
```

Vim会输出“3”，即是字符串“foo”的长度。现在运行下面的命令:

```
:echom len("foo")
```

Vim这次也是输出“3”。用len和strlen对字符串操作的结果是一样的，我们后面会继续讨论len函数。

字符串切割

运行下面的命令（注意这里用的是echo而不是echom）：

```
:echo split("one two three")
```

Vim会输出“['one','two','three']”。spilt函数将字符串切割成一个字符串列表。现在不必考虑列表，我们后面会讨论它的。

你可以显式高速vim来用什么字符切割，而不是用默认的空格。运行下面的命令：

```
:echo split("one,two,three",",")
```

这次vim还是会输出“['one','two','three']”，因为第二参数高速vim来用逗号进行切分。

连接

你不仅仅可以切割字符串，你还可以对它们进行连接的操作。运行下面的命令：

```
:echo join(["foo","bar"],"...")
```

Vim会输出“foo...bar”，暂时不用担心上面的语法。

spilt可以和join可以配对起来达到很不错的效果。运行下面的命令：

```
:echo join(split("foo,bar"),";")
```

Vim会输出“foo;bar”，首先我们把字符串"foo bar"切割成一个列表，然后用分号来连接它们。

大小写转换

Vim有两个函数用来改变字符串的大小写，运行下面的命令：

```
:echom tolower("Foo")
:echom toupper("Foo")
```

Vim会输出“foo”和“Foo”，这个很容易就能够理解了。

在很多语言里（例如Python），在进行一个大小写不敏感的比较之前，一个通用的规则就是先将字符串转换成小写的形式。但是在vim里不需要进行这样的操作，因为vim有大小写敏感比较符。如果你不记得这个的话，重新阅读以下比较运算那一章。

不过，究竟使用tolower，还是==#还是用==?，这都取决于你自己。对于这个vimscript社区里没有一个强制的约定。随便选择一个你觉得合适的，保持使用它。

练习

- 阅读:help functions浏览一下vim里内置的提到字符串的函数。用/来进行搜索（注意，vim的帮助文档也可以像其他文件一样进行操作）。这里有很多函数，所以你不需要挨个的去读，只要记住当你需要的时候可以在这里面进行查阅。

execute命令

execute命令是用来把一个字符串当作vimscript来进行执行的。运行下面的命令: `:execute "echom 'Hello, world!'"`

Vim会把echom 'hello, world!'作为命令进行执行，即是输出字符串并且打下消息日志。Execute命令的功能非常强大，因为你可以把任意字符串当作命令来执行。

我们来看看一个更加有用的例子。首先在vim里打开一个文件，然后用:edit "foo.txt"，在当前窗口打开另外一个缓冲区。现在运行下面的命令： `:execute "rightbelow vsplit " . bufname("#")`

vim会在第二个文件的右边分出一个窗口，并在里面打开第一个文件。究竟是发生了什么呢？

首先，vim会把“rightbelow vsplit”和bufname("#")的执行结果连接起来组成一个命令。

后面我们会讨论bufname这个函数的，现在只要知道它会返回之前一个文件的路径就可以了。如果你想验证的话，你可以用echom来打印出它的执行结果。

一旦bufname函数被执行了，vim就会把把它的结果和前面的字符串拼接成一个类似于“rightbelow vsplit bar.txt”的字符串。然后execute会执行这个vimscript命令，从而就会在一个窗口里打开这个文件。

execute危险吗？

在大部分的编程语言里，用这种“eval”结构来直接执行命令的语是不推荐使用的。但是，在vimscript里确不是这样。主要有以下两个原因。

首先，vim只会从一个人那里得到指令：也就是用户。如果用户想用execute命令来搞破坏的话，那就随他的便了，那是他自己的电脑。

和其他的时常会从不可信的用户那里取得输入的编程语言相比，vim的环境比较特殊，一些基本的安全问题都不很常见。

第二个原因是vimscript有些时候要用一些晦涩带技巧性的语法，execute经常是最简单，最直接的可以完成任务的方式。在其他编程语言里“eval”结构并不会省下多少编码，但是在Vimscript里可以很方便的把多行命令写在一行里。

练习

- 浏览:help execute 看看哪些事情可以用execute来做，哪些又是不能的。不要陷入的太深，我们后面会继续讨论它的。
- 阅读:help leftabove，:help rightbelow，:help :split，和:help :vsplit（注意后面两个命令里的分号）
- 在你的vimrc文件里添加一个映射用来在你选择的位置（水平/竖直，上/下/左/右）打开之前的文件。

Execute Normal! 命令

现在为止，我们已经见过了execute和normal！，我们可以讨论一个常用的Vimsctipt技巧。运行下面的命令：

```
:execute "normal! gg/foo\<cr>dd"
```

这个命令会让光标移动到文本的开始处，查找第一次出现“foo”的文本行，然后删除它。

之前我们曾经始终用normal!命令来执行一个搜索的命令，但是没法输入回车键来进行搜索。把normal!和execute结合起来就可以修复这个问题。

execute可以让我们构建一些很神奇的命令，所以你可以用vim的字符串里的转义字符来生成我们需要的非打印字符。运行下面的命令： `:execute "normal! mqA;\<esc>` q"`

这个命令的作用是什么呢？我们来把它分解一下。

:execute "normal ! ..." 用来执行命令，就像在normal模式下执行一样，忽略所有的映射，并且把转义字符替换为它原本的意义。

- mq： 把当前的位置保存在书签“q”里。
- A:移动到当前文本行的结尾处，然后再最后一个字符后面进入insert模式。
- ;：现在我们在insert模式，这是在最后一个字符后面插入分号 i- \：这个带有转义符的字符串，会被解析成相当于按下了escape键，这样就会让我们退出insert模式。
- `q：返回到书签“q”的位置。

  这个命令看起来有点吓人，但是却很有用：它在当前行的后面添加一个分号，并且保持光标在原来的位置不动。这个映射在你用Javascript，C语言或者其他以分号作为语句的结尾的语言编写代码的时候，如果忘了加上分号的时候会非常有用的。

练习

- 再次阅读:help expr-quote（你之前已经看过了）来复习一下怎么用execute来给normal!传递特殊字符。
- 在你阅读下一章前，先放下本书，吃个三明治，或者喝杯咖啡，如果有个宠物的话，你也可以喂喂它，然后休息一会。

基本正则表达式

Vim是一个文本编辑器，也就是说你的大部分Vimscript代码的作用都是操作文本。Vim对正则表达式有着很强的支持，但是和其他的功能一样，这里也有一些陷阱。

把下面的文本输入到缓冲区：

```
max = 10

print "Starting"

for i in range(max):
     print "Counter:",i

print "Done"
```

这就是我们要用来对Vimscript进行正则实验的文本。

我会假设你已经有了基本的正则知识。如果你不知道正则表达式的话，你需要暂时停住，然后去阅读Zed Shaw写的《Learn Regex the Hard Way》，读完之后再继续回到这里。

高亮文本

在我们开始之前，先需要打开高亮搜索的设置，这样我们就能够很清楚的看到自己在干什么。运行下面的命令：

```
:set hlsearch incsearch
```

hlsearch用来告诉Vim去高亮文件里所有匹配的文本，incsearch告诉vim在你输入搜索表达式的时候进行增量匹配，也就是在你还在输入的时候就开始进行匹配。

搜索

把光标放在文件的开始处，然后运行下面的命令：

```
/print
```

在你输入每个字符的时候，vim就从第一行开始进行高亮显示了。当你输入回车键来执行搜索的时候，所有的匹配到“pring”的实例都会被高亮显示，然后你的光标会被移动到接下来第一个匹配上的文本。

现在运行下面的命令：

```
:execute "normal!  gg/print \"
```

首先，光标会移动到文本的开始处，然后搜索“print”，并让光标停留在第一个匹配的地方。这个是通过:execute "normal! ..."来实现的，这个我们之前就见过的。

如果要移动到第二个匹配的结果上去，只需要在后面再加上命令即可。运行下面的命令：

```
:execute "normal! gg/print\n"
```

Vim会把光标移动到第二个匹配的文本上去（并且所有匹配的文本都高亮显示）。

现在我们再反向操作，运行下面的命令：

```
:execute "normal! G?print\"
```

这次我们用G来移动到文本的末尾，然后用?来进行反向搜索。

这些搜索的命令相信你已经很熟悉了——我们曾多次用它们来讲解:execute "normal! ...的用法，因为它可以让你在Vimscript代码里做任何可以在vim里做的事情。

魔法

/和?命令的参数是正则表达式，不仅仅只是字符串。运行下面的命令：

```
:execute "normal! gg/for .+ in. +:\"
```

这次vim会抱怨说找不到要搜索的字符串！我前面告诉过你Vim的搜索是支持正则表达式的，但是是什么原因导致搜索不出来呢？运行下面的命令：

```
:execute "normal! gg/for .\\+ in .\\+:\"
```

这次vim会高亮for语句，这正是我们所期望的。花几分钟的时间来看看想想这个命令和之前到底有什么变化呢。记住execute要的只是一个字符串。

我们为什么要像上面那样来写正则表达式的命令呢，主要有两个原因：

- 首先，execute的参数是字符串，所以我们需要用双斜杠来表示单斜杠。
- 其次，vim有4种不同的模式来解析正则表达式！默认的模式是需要在+号前面带斜杠，来表示后面有一个或者多个字符，而不是表示字符“+”。

你可以通过只在vim里用更简单的方式来使用它。运行下面的命令：

```
/print .\+
```

你可以看到+也起作用了。双斜杠只是在需要把把正则以字符串的方式传递给execute的时候使用。

字符串常量

我们之前已经提到了，vim运行你用单引号来定义一个“字符串常量”，即是所有的字符都当作原形处理。例如，字符串“a\n\b”就是代表四个字符。

那么在正则表达式里可以用字符串常量来避免要输入双斜杠吗？仔细考虑一下，这个问题的答案比我们想象的要复杂那么一点点。

运行下面的命令（注意单引号和单反斜杠）：

```
:execute 'normal! gg/for .\+ in .\+:\'
```

Vim会把光标移动到文本的开始处，但是并不会移动到第一个匹配的文本。这是你所期望的吗？

这个命令不会起作用是因为我们需要\被转义成一个真正的换行符来让搜索的命令执行。因为我们用的是字符串常量，这个字符串就相当如在vim里直接输入“/for .+ in: .+:\”这个字符串，显然这不是我们想要的。

但是，并不是一点都不能用的。还记得vim里是允许进行字符串拼接的吗，所以对于较长的命令，我们可以把它分割成更加容易阅读的部分。运行下面的命令：

```
:execute "normal! gg". '/for .\+ in .\+:' . "\"
```

这个命令先把三个字符串拼接起来，然后一起传给execute来执行，这样就允许我们把正则部分的字符串用字符串常量来表示，而一般字符串这直接表示。

更加不可思议的

你也许还在想Vimscript的4种不同模式的解析方式，以及它会同你在Python，Perl，Ruby等语言里学到的正则的差别。如果你想知道的比较全面的话，你可以去阅读以下文档，如果只是想简单了解的话，就借着往下读。

运行下面的命令：

```
:execute "normal! gg" .'/\vfor .+ in .+:' . "\"
```

这次我们再次把命令后面的部分分割成字符串常量，并且这次我们的正则则是以“\v”开头的。这个会告诉vim来用它的“very magic”模式来解析，这个模式和你在其他编程语言里用到的很相似。

如果你的正则是以“\v”开始的话，那么你就再也不需要担心vim的其他三种正则表达式的模式了。

练习

- 仔细阅读:help magic。
- 阅读:help pattern-overview，看看vim支持那些正则表达式。在字元的地方停止。
- 阅读:help match。试试:match Error /\v../命令。
- 编辑你的vimrc文件，添加一个映射，用它来把行尾的空格作为错误来高亮显示。一个不错的选择是用w作为快捷键。
- 添加另外一个映射用来删除上面匹配到的文本。
- 添加一个normal模式下的影视，使得你在开始搜索的时候自动帮你输入\v。如果你忘记了怎么实现，你只要记住vim的映射很简单，你只要告诉它当你在使用映射的时候是需要输入什么就好了。
- 把hlsearch和incsearch选项添加到你的vimrc文本里，根据你的需要去设置它们的值。
- 阅读:help nohlsearch。记住，这是一个命令，并且不是hlsearch的关闭状态！
- 添加一个映射，实现停止上一个搜索的高亮显示。

实例学习：Grep操作符之一

从本章开始，我们将会来构造一些很复杂的Vimscript。我们将会讨论一些我们之前没有看到过的东西，并且把它们和我们之前学习的东西结合起来使用。

在你学习这些例子的时候，如果碰到不熟悉的东西，记得用:help命令来查看一下。如果你没有完全理解所有的东西的话，你就不会学到很多东西的。

Grep

如果你从来没有用过:grep，你需要花几分钟时间来阅读一下:help :grep和:help :make。如果你没有用过快速补全窗口，那么也阅读一下:help quickfix-window。

简单的来说，:grep ...调用外部命令来处理你输入的参数，解析返回结果，并填充在快速补全窗口里，以方便在vim里使用。

在我们的例子里，为了方便调用，我们会添加一个“grep操作符”，使得你能够在任何vim的内置动作来选择你要搜索的文本。

用法

当你在写一段不简单的Vimscript时，你首先要考虑的事情是：“这个功能会被怎么使用呢？”。试着去想出一个对你以及你的代码的使用者而言更加流畅，更加舒服，更加直观的方式。

在这个例子里，我会帮你做上门那一步：

- 我们会创建一个“grep操作符”，并且和g进行绑定。
- 它会像w以及i{这些Vim操作符一样，会接受一个动作。
- 它会立马执行这个搜索，并且把结果显示在快速补全窗口里。
- 它不会跳到第一个匹配的文本上去，因为那不一定是我们要的操作。

一些你可能会使用这个功能的例子有：

- giw：搜索当前光标下的单词
- giW：搜索当前光标下的WORD
- gi' ：搜索光标当前所在单引号里的内容
- viweg：在visual模式下选择一个单词，并且把选中的区间扩展到下一个单词的结尾处，然后搜索选中的文本。

  当然，还有其他很多可以用到这个的地方。看起来实现这些功能需要写很多代码，其实我们只要实现这个操作符，其他的东西，vim会帮我们处理的。

一个基本的框架

当你需要写一些棘手的Vimscript时，一个比较好的方式就是，先简化你的目标，并且实现它来让你对你的最终目标有个基本的了解。

让我们把目标简化为“创建一个映射来搜索当前光标下的单词”。这个也很有用，并且也比较简单，这样我们就能够更快的运行我们的代码。我们会暂时把这个命令映射为g。

我们会先从这个脚本的基本模型开始，然后再慢慢完善它。运行下面的命令：

```
:nnoremap g :grep -R something .
```

如果你阅读过:help grep的话，那么这个命令就很容易去理解了。我们之前已经看过很多映射了，这个就没什么新的东西了。

很显然这个还没有完成我们的功能，所以我们需要继续完善它以至于能够达到我们简化的目标。

搜索的内容

首先，我们要搜索的是当前光标下的单词，而非是someting。运行下面的命令：

```
:nnoremap g :grep -R  .
```

现在试试它。是vim的一个特殊标识符，它会在运行命令之前被vim替换成光标下的单词。

你可以用来获取当前光标下的文本串。运行下面的命令:

```
:nnoremap g :grep -R  .
```

现在把你的光标放在类似于“foo-bar”的文本上，再试试上面的命令。vim会搜索“foo-bar”，而不只是这个文本的一部分。

但是这里还有另外一个问题，如果选择的文本里有shell的关键词的话，vim也是会传过去的，但是这样会出问题的（比较坏的情况是做一些很可怕的事情）。

你可以去试试这样做，以加深印象。在一个文件里输入foo;ls，保持光标在这个文本串上，然后运行上面那个映射。这时，grep命令会运行失败，但是vim会同时运行一个ls命令。很显然，如果是其他比ls更加危险的命令的话，那就比较可怕了。

我们可以通过给grep的参数加上引号来解决这个问题。运行下面的命令：

```
:nnoremap g :grep -R '' .
```

大部分的shell也会把单引号里的字符串当作常量来处理，所以我们的映射就更加健壮了。

转义**shell**命令的参数

但是，对于搜索的内容，这里还存在一个问题。试着在“that's”文本上使用这个映射。它不起作用，因为搜索文本里的单引号对grep命令的单引号造成了影响。

为了解决这个问题，我们可以用Vim的shellescape函数。阅读:help escape()和:help shellescape()来看看它是如何工作的（这个非常简单）。

因为shellescape()函数只对vim字符串有效，所以我们需要动态拼接命令和execute。首先运行下面的命令来把:grep映射成:execute "..."的形式：

```
:nnoremap g :execute "grep -R '' ."
```

试试上面的命令，保证它还能够正常工作。如果不行的话，找到输错的地方并进行纠正。然后运行下面的命令，这个命令里用shellescape函数来修复搜索内容上的问题：

```
:nonoremap g :execute "grep -R " . shellescape("") . " ."
```

试着对一个正常的单词例如“foo”执行上面的命令，它很好地起了作用。再试试一个带引号的单词，例如“that's”。它又失效了，到底是怎么一回事呢？

这是因为Vim会在特殊字符串被替换之前就先进行了shellescape()函数。所以Vim直接对字符串“”进行了转义（只是往里面添

加单引号来进行转义），然后把它和grep命令串拼接起来。

你可以通过下面的命令来验证上面的说法：

```
:echom shellescape("")
```

Vim会输出"。注意两边的双引号也是字符串的一部分，因为vim这样做是为了让它用做一个shell命令的参数。

可以用expand()函数来修复它，expand函数可以在字符串传给shellescape之前，强制把它转换成它实际上是表示的文本。

我们可以先看看expand函数的功能。把你的光标放在一个带引号的字符串串上，例如“that's”，然后运行下面的命令：

```
:echom expand("")
```

Vim会输出“that's”，因为expand("")会把当前光标所在位置的单词作为vim字符串返回。现在把shellescape加入到我们的命令里：

```
:echom shellescape(expand(""))
```

这次Vim会输出'that'\''s'。这个看起来有点复杂，相信你不愿意去理睬shell的复杂语法。现在，不用担心这个，只要相信vim从expand里拿到字符串，并正确的进行了转义即可。

现在我们知道了怎么取得当前光标下的文本，并且进行转移了。现在只需要把它和我们的影射结合起来就可以了！运行下面的命令：

```
:nnoremap g :execute "grep -R " . shellescape(expand("")) . " ."
```

再试试看。这次我们的命令不会再因为我们搜索的文本里包含特殊字符而崩溃了。

这个通过一个小的Vimscript，来慢慢进行转换成接近你目标的版本的方法会对你很有用的。

清理一些小问题

这里还有一些小问题我们需要关注。首先，我们说过，我们不不需要自动跳到第一个匹配的文本上去，我们可以用grep!代替grep来实现它。运行下面的命令：

```
:nnoremap g :execute "grep! -R " . shellescape(expand("")) . " ."
```

试试这个命令，不过什么都不会出现。Vim已经在快速补全窗口里填充了结果，但是我们还没有打开它。运行下面的命令：

```
:nnoremap g :execute "grep -R " . shellescape(expand("")) . " .":copen
```

在试试上面的命令，你可以看到vim会打开快速补全窗口，并且里面有搜索的结果。我们所做的只是在映射的末尾加上:copen。

作为最后一步的工作，我们需要清除在搜索时，vim打印出的所有grep命令的输出。运行下面的命令：

```
:nnoremap g :silent execute "grep! -R " . shellescape(expand("")) . " ."
```

现在，我们的工作都完成了，现在试一试。silient命令的功能是只运行它后面的命令，但是忽略它所有应该输出的信息。

练习

- 把刚刚创建的影射添加到你的vimrc文本里。
- 如果你之前没有阅读:help :grep的话，阅读一下它。
- 阅读:help cnext和:help cprevious。在使用了上面的映射之后，试试它们。
- 添加:cnext和cprevious的映射，以便于快速切换搜索结果。
- 阅读:help expand。
- 阅读:help copen。
- 给映射里的:copen命令添加一个高度，是得每次它打开的时候都是你觉得合适的高度。
- 阅读:help silent。

实例分析：**grep**操作符，第二部分

现在我们已经有了最终方案的基本雏形了，现在是时候来让它变功能更加强大了。

记住我们最开始的目标是创建一个grep操作符。为了达到这个目标，我们还有很多需要去做的，但是这里我们还是要和上一章节一样，先从一些简单的地方开始，然后把它变成我们所要的结果。

在开始之前，我们先要把之前在vimrc文件里添加的映射给删除掉，因为接下来我们还是要用相同的按键来做映射。

创建一个文件

创建一个操作符需要很多的命令，如果手动去输入的话，就比较麻烦了。你也可以把这些命令添加到你的vimrc文件里，但是这里我们用一个单独的文件来存放这些命令。用单独的文件的话就更便于维护了。

首先，找到你的Vim plugin目录。在Linux或者OS X系统下，这个目录在~/.vim/plugin。如果是windows系统，它会再你的home目录下的vimfiles里（如果你不清楚的话，可以用在vim里用:echo $HOME来查看）。如果这个目录不存在，你可以直接创建一个。

在plugin目录里创建一个grep-operator.vim的文件。在这个文件里会放关于这个新操作符的命令。当你编辑完这个文件后，可以用:source %来重新加载它，使得新编辑的代码生效。这个文件也会在你每次打开vim的时候自动加载，就像~/.vimrc文件一样。

记住每次加载之前要先写入文件，才能看得到效果!

基本框架

要写一个新的vim操作符，你需要从两个组件开始：一个函数和一个映射。首先把下面的代码加入到grep-operator.vim里：

```
nnoremap g :set operatorfunc=GrepOperatorg@

function! GrepOperator(type)
     echom "Test"
endfunction
```

写入该文件，并且用:source %来加载它。通过按键giw来触发“单词内搜索”的功能。Vim会在接受到iw动作的时候输出“Test”，这说明我们的基本框架已经搭建好了。

这个函数很简便，并且没有什么是我们以前没有接触到的，但是这个映射是有一点点复杂的。首先我们operatorfunc选项设置成我们的函数，然后然后g@,这样会把这个函数当作操作符来调用。这个看起来有点晕，但是实际上就是这么运行的。

现在可以把这个映射当作一个很盒子来看待。后面你可以仔细研究一下相关的文档。

Visual模式

我们把操作符加入到normal模式里了，但是也想在visual模式下去使用它。这之前的映射下面再添加一个映射：

```
vnoremap g :call GrepOperator(visualmode())
```

写入并且加载这个文件，现在在visual模式下选中一些文本，并且按g。什么都没有发生，但是vim输出了“Test”，所以这说明我们的函数被调用了。

我们之前已经看到过很多次，但是从来没有解释它的作用。试着在visual模式下选中一些文本，然后输入:。vim这时会显示一个命令行，但是和一般情况不同的是它会在命令行的开始处填入'<,'>。

Vim这样做是为了方便你，插入这些文本的作用是为了让你将要运行的命令作用在选择的文本上。但是在这里，这个功能不起任何作用。我们用来告诉vim删除从光标到行首的文笔，来删除这些文本。这样就会只剩下一个:，接下来就可以用call调用函数了。

这里的call GrepOperator()和之前我们所见到的函数调用一样，不过这里我们用了一个visualmode()函数来作为参数，这个之前是没有见过的。这是一个vim的内置函数，它会返回一个单字符的字符串，来代表visual模式的上一次输入：'v'代表characterwise，"V"代表linewise，ctrl-v代表blockwise。

动作的种类

我们之前定义的函数会接收一个type参数。我们知道当在visual模式下的时候，这个参数是visualmode()函数的返回值，那么在normal模式下作为一个operator运行时的参数又是什么呢?

编辑函数，使得和下面的内容相同：

```
nnoremap g :set operatorfunc=GrepOperatorg@
vnoremap g :set operatorfunc=GrepOperator

function! GrepOperator(type)
     echom a:type
endfunction
```

加载这个文件，通过不同的方式来试试这个函数。一些示例的输出如下面所示：

- 按viw 会输出v，因为我们在chacterwise模式
- 按Vjj 会输出V，因为我们在linewise模式
- 按giw输出char，因为我们用了一个characterwise动作。
- 按gG 输出line，因为我们对这个操作符用了一个linewise动作。

现在我们知道怎么来分辨不同的动作类型了，这个在我们选择搜索的文本的时候是很重要的。

复制文本

我们的函数需要能够操作用户想要去搜索的文本，做简单的方式就是去复制它。编辑函数，使得它的内容如下：

```
nnoremap g :set operatorfunc=GrepOperatorg@
vnoremap g :call GrepOperator

function! GrepOperator(type)
     if a:type ==# 'v'
          execute "normal! `v`y"
     elseif a:type ==# 'char'
          execute "normal! `[v`]y"
     else
          return
     endif

     echom @@
endfunction
```

现在这个函数里多了很多新的内容。试试类似于giw,g2e和vi()g等命令。每次vim会输出当前动作所包含的文本，很显然，我们的功能在慢慢完善。

我们来分解一下上面的新代码。首先我们有一个if语句，它会根据a:type的不同值执行不同的命令。如果当前的type是v，也就是说这个函数是从characterwise的visual模式下调用的，所以我们就通过命令来复制visual模式下选中的文本。

注意我们用的是大小写敏感的比较符号==#。如果我们只是用==来进行比较，并且用户设置了ignorecase选项的话，就会导致'V'也被匹配上，这不是我们想要的结果。防御式编程！

第二个if语句会在操作符在visual下用一个characterwise动作调用的时候执行。

最后一个分支直接返回了。我们显示地忽略了linewise/blockwise visual模式和linewise/blockwise动作。Grep默认不支持跨行的搜索，所以在搜索的文本里带入一个换行符是没有意义的。

上面的两个if语句都执行了一个normal!命令，它们做了两件事：

- 在visual模式下选择我们要的文本范围：
  - 移动到文本范围开始的标志处
  - 进入characterwise visual模式
  - 移动到文本范围结尾的标志处
- 复制选中的文本

暂时不用考虑上面用到的特殊标志。在完成了本章的练习后，你会明白为什么要用不同的命令来做同样的操作。

函数的最后一行输出变量@@。记住，以@开始的变量都是寄存器。@@是无名寄存器，它就是在没有指定寄存器进行复制和删除操作时，vim存放文本的寄存器。

简而言之：我们选中了要搜索的文本，复制它，并且粘贴它。

转义搜索文本

现在我们已经有了我们需要的文本，接下来我们需要像在前面一章一样，对它进行转义。把echom命令修改成下面所示：

```
nnoremap g :set operatorfunc=GrepOperatorg@
nnoremap g :call GrepOperator

function! GrepOperator(type)
       if a:type ==# 'v'
           execute "normal! `v`y"
      elseif a:type ==# 'char'
           execute "normal! `[v`]y"
      else
           return
      endif

      echom shellescape(@@)
endfunction
```

写入文件，并加载它。试着在visual模式下选择一段带有特殊字符的文笔，然后输入g>。Vim会输出选中文本转义后的内容，以便作为参数传递给shell命令。

运行Grep

现在我们就只差添加grep!命令了，加上它就可以进行搜索的动作了。替换echom命令：

```
nnoremap g :set operatorfunc=GrepOperatorg@
vnoremap g :call GrepOperator(visualmode())

function! GrepOperator(type)
    if a:type ==# 'v'
        normal! `y
    elseif a:type ==# 'char'
        normal! `[v`]y
    else
        return
    endif

    silent execute "grep! -R " . shellescape(@@) . " ."
    copenendfunction
```

现在看起来就有点熟悉了。我们只是在后面执行了上一章的“silenct execute "grep! ...”命令。现在这个命令的可读性更高了，

因为我们不再是把所有的内容都堆在一行里了。

写入文件，并加载。试试这个操作符，并且享受一下你的劳动成果。

因为我们已经定义了一个新的操作符，所以我们可以用不同的方式来使用它，例如：

```
viwg: 在visual模式下选择一个单词，并且搜索它

g4w: 搜索接下来的四个单词

gt;: 搜索直到分号

gi[: 搜索方括号里的内容
```

这个强调了vim里最有用的东西：它的编辑命令就像一门语言。当你添加一个新的动词，它就能自动地和大部分的已存的名词和形容词结合起来。

练习

- Read :help visualmode().
- Read :help c_ctrl-u.
- Read :help operatorfunc.
- Read :help map-operator.

实例分析：**Grep**操作符，第三部分

我们的新grep操作符工作的很好，但是有些Vimscript可以改善一下，使得我的操作符更加体贴，更加方便用户去使用。我们可以再做两件事情来让我们的操作符在vim系统里工作的更好。

保存寄存器

通过把文本保存到未名的寄存器里，我们把之前存在里面的内容给覆盖了。

这个对于我们的用户而言，是很不友好的，所以我们需要在复制文本并保存到寄存器里之前把寄存器里的内容先存下来，然后再我们的操作完成之后进行保存。编辑之前的代码，使得和下面相似：

```
nnoremap g :set operatorfunc=GrepOperatorg@
vnoremap g :call GrepOperator(visualmode())

function! GrepOperator(type)
    let saved_unnamed_register = @@

    if a:type ==# 'v'
        normal! `y
    elseif a:type ==# 'char'
        normal! `[v`]y
    else
        return
    endif

    silent execute "grep! -R " . shellescape(@@) . " ."
    copen

    let @@ = saved_unnamed_register
endfunction
```

我们在函数的开始处和结束处分别加了一个let语句。第一个是用来保存@@里的内容，第二个则是恢复它里面的内容。

写入脚本，并且加载文件。检查它是否正常工作，首先复制一些文本，然后按giw来触发我们的操作符，然后按p来粘贴你之前复制的文本。

在写Vim插件的时候，你都要保存和恢复你的代码修改的任何用户配置以及寄存器的内容。

命名空间

我们的脚本在全局命令空间里创建了一个GrepOperator。这个或许没什么大不了的，但是当你在写vimscript的时候保证安全绝对比时候后悔要强。

我们可以通过对代码稍微做一些调整来防止污染全局命令空间。编辑代码，使得和下面相似：

```
nnoremap g :set operatorfunc=GrepOperatorg@
vnoremap g :call GrepOperator(visualmode())

function! s:GrepOperator(type)
    let saved_unnamed_register = @@

    if a:type ==# 'v'
        normal! `y
    elseif a:type ==# 'char'
        normal! `[v`]y
    else
        return
    endif

    silent execute "grep! -R " . shellescape(@@) . " ."
    copen

    let @@ = saved_unnamed_register
```

```
endfunction
```

代码的前三行都有了一点变化。首先，我们用:s来修饰函数名称，这样就可以把它放在当前脚本的命名空间里了。

同时我们也修改了映射，我们在GrepOperator函数名的前面加上了,，这样就可以保证它们可以找到对应的函数了。如果我们不这样做的话，vim就会在全局命名空间里来找这个函数，这样肯定是不行的。

庆祝一下，我们的grep-operator.vim脚本现在非常有用了，现在可以把他当作vim内置的操作符来使用了。

练习

- 阅读:help
- 给自己买一些零食，这是你该得的。

列表

我们对单独的变量进行了很多讨论，但是没有讨论聚合类型的变量。Vim有两种主要的聚合类型的变量，我们现在先看看第一种：列表。

Vimscript列表里都是有序的，混合类型的元素。运行下面的命令：

```
:echo ['foo','3','bar']
```

Vim会输出上面的列表。当然，列表也是可以嵌套的，运行下面的命令：

```
:echo ['foo',[3,'bar']]
```

Vim也会输出列表。

索引

Vimscript的列表是从零开始索引的，你可以用一般的方法来取得列表里的元素。运行下面的命令：

```
:echo [0,[1,2]][1]
```

Vim会输出“[1,2]”。你也可以像Python里面一样从列表的后面开始进行索引。运行下面的命令：

```
:echo [0,[1,2]][-2]
```

Vim会输出0。-1索引的是倒数第一个元素，-2索引的是倒数第二个元素。

数组切分

Vim里的列表是可以被切分的。这个和Python里的也很相似，但是它并不一定每次的表现都是一样的。运行下面的命令：

```
:echo ['a','b','c','d','e'][0:2]
```

vim会输出“['a','b','c']”（索引分别为0，1，2的元素）。你也可以超出索引，并且是安全的。运行下面的命令：

```
:echo ['a','b','c','d','e'][0:100000]
```

Vim会输出整个列表。

切割的索引也可以是负数。运行下面的命令：

```
:echo ['a','b','c','d','e'][-2:-1]
```

Vim会输出['d','e']（索引分别为-2和-1的元素）。

当你在分割的时候，你可以省略第一个索引表示从列表的第一个元素开始，也可以省略第二个索引表示到列表的末尾截止。运行下面的命令：

```
:echo ['a','b','c','d','e'][:1]
:echo ['a','b','c','d','e'][3:]
```

Vim会输出“['a','b']”和“['d','e']”。

和Python一样，Vimscript也允许你对字符串进行索引。运行下面的命令：

```
:echo "abcd"[0:2]
```

Vim会输出“abc”。

拼接

你可以用+来拼接Vim列表。运行下面的命令：

```
:echo ['a','b']+['c']
```

当然，vim还是毫无意外地输出“['a','b','c']”。这儿也没什么好讲的——Vimscript里的列表和其他的功能相比还是比较正常的。

列表函数

Vim里有很多内置的用于操作列表的函数。运行下面的命令：

```
:let foo = ['a']
:call add(foo,'b')
:echo foo
```

Vim会就地改变列表foo，在它的末尾添加'b'，然后会输出“['a','b']”。现在运行下面的命令：

```
:echo len(foo)
```

Vim输出“2”，也即是列表的长度。运行下面的命令：

```
:echo get(foo,0,'default')
:echo get(foo,100,'default')
```

Vim会输出“a”和“default”。get函数会返回指定位置的元素，如果指定的索引超出列表的范围，就会返回指定的默认值。

运行下面的命令：

```

```

```
:echo index(foo,'b')
:echo index(foo,'nope')
```

Vim会输出1和-1.index函数返回指定的元素在列表中第一次出现的位置的索引，如果指定的元素不存在，则返回-1。

现在，再运行下面的命令：

```
:echo join(foo)
:echo join(foo,'---')
:echo join([1,2,3],'')
```

Vim会输出“a b”，“a---b”，和“123”。join函数会把指定列表里的元素拼接成一个字符串，各个元素之间用给定的字符串分隔（如果没有指定的话，默认以空格分隔），如果需要的话，会强制把每个元素转换成字符串。

运行下面的命令：

```
:call reverse(foo)
:echo foo
:call reverse(foo)
:echo foo
```

Vim会先输出“['b','a']”，然后输出“['a','b']”。reverse函数会倒转给定的列表。

练习

- 阅读 :help List.阅读所有的内容，注意大写的L。
- 阅读 :help add().
- 阅读 :help len().
- 阅读 :help get().
- 阅读 :help index().
- 阅读 :help join().
- 阅读 :help reverse().
- 浏览 :help functions，看看那些本章没有提到的list操作的函数。 运行:match Keyword /\clist/ 来高亮小写的list，以便于找到要看的内容。

循环

在我们已经过了35个章节才来讲解循环，这点可能会让你觉得比较奇怪。这是因为vim提供了很多其他的方式来对文本进行操作（例如normal!），这样就使得循环在vim里没有像其他编程语言里显得那么重要了。

即使是这样，但是你肯定还是会需要这个功能的，所以我们会看看vim里主要的两种循环方式。

For循环

我们要看的第一个循环是for循环。如果你比较习惯于Java，C或者Javascript的for循环的话，那么这里的for循环看起来就有些别扭了，不过这里的看起来更加高雅。运行下面的命令：

```
:let c = 0
:for i in [1,2,3,4]
:     let c += i
:endfor

:echom c
```

Vim会输出“10”，即是列表里的元素的和。Vimscript的for循环可以迭代列表以及后面会提到的字典。

vim里没有像C语言一样的 for (int i = 0;i < foo; i++)的循环方式。这个可能刚开始看起来的时候比较麻烦，但是你用习惯了的话就不会去想它

While循环

Vim也支持传统的while循环。运行下面的命令：

```
:let c = 1
:let total = 0
:while c <= 4
:     let total += c
:     let c+= 1
:endwhile

:echom total
```

这次Vim还是会输出“10”。这个循环对任何一个有编程经验的人都是熟悉的，所以这里我们就不讲解了。这个循环你很少会用到的。你只需要在碰到的时

练习

- 阅读:help for
- 阅读:help while

字典

我们最后要讨论的一个Vimscript变量是字典。Vimscript里的字典和Python里的dicts，Ruby里的hashes，已经javascript里的object非常相似。

字典都是通过一对大括号来创建的。字典里的值都是任意的，不过键的值基本上都是字符串。你不会认为大部分情况都是正常的，对吗？

运行下面的命令：

```
:echo {'a' : 1,100:'foo'}
```

Vim会输出{'a': 1, '100': 'foo'}，你可以看到Vimscript确实把键值强制转换成字符串了，而值没有进行转换。

Vimscript避免了Javascript愚蠢的规范，它要求你在字典的末尾的元素后面加上逗号。运行下面的命令：

```
:echo {'a': 1, 100: 'foo',}
```

这次Vim还是输出了{'a': 1, '100': 'foo'}。你要记住在字典的末尾加上逗号，尤其是你的定义跨多行的时候，因为这样会在你添加新元素的时候减少出错的概率。

索引

在一个字典中查找一个key，你可以使用和大部分语言差不多的语法。运行下面的命令：

```
:echo {'a': 1, '100': 'foo',}['a']
```

Vim会输出1。试试一个非字符串的索引：

```
:echo {'a': 1, '100': 'foo',}[100]
```

Vim会在进行查找前，先把非字符串的索引转换成字符串，这样就可以进行查找了，因为所有的键值都是字符串。

当key只由字母，数字以及下划线组成的时候，Vimscript也支持Javascript的“.”操作符进行查找。运行下面的命令：

```
:echo {'a': 1, '100': 'foo',}.a
:echo {'a': 1, '100': 'foo',}.100
```

对于上面两个场景，Vim都会输出正确的元素。怎么选择索引的方式，取决于你个人的喜好。

赋值和添加元素

添加元素的操作和复制的操作一样。运行下面的命令：

```
:let foo = {'a', 1}
:let foo.a = 100
:let foo.b = 200
```

```
:echo foo
```

Vim输出{'a': 100, 'b': 200}，这样就表明了复制和添加元素的操作是一样的。

删除元素

Vimscript里有两种方式来从字典里删除元素。运行下面的命令：

```
:let test = remove(foo,'a')
:unlet foo.b
:echo foo
:echo test
```

Vim会输出{}和100。remove函数会把指定的键值所对应的条目删除掉，并返回删除的条目对应的值。unlet命令也可以删除字典里的条目，但是你没法使用它的值。

你不能从字典里删除不存在的条目。运行下面的命令：

```
:unlet foo["asdf"]
```

Vim会抛出错误。

对于remove和unlet的选择也是依据个人的喜好来的。如果非要做出一个选择的话，我推荐只使用remove，因为它比unlet更加灵活。remove可以做unlet所能做的所有功能，但是反过来就不成立了，如果你要保持一致的话，那就只用remove。

字典函数

和列表一样，vim里有很多内置的函数用来操作字典的。运行下面的命令：

```
:echom get({'a':100},'a','default'}
:echom get({'a':100},'b','default'}
```

Vim会输出100和default，这个和list的函数一致。

你也可以通过函数来检测一个指定的键值是否在字典里。运行下面的命令：

```
:echom has_key({'a':100},'a')
:echom has_key({'a':100},'a')
```

Vim会输出1和0。记住在vim里0代表假，其他的数字代表真。

你可以通过items函数来把字典里的所有条目都取出来，运行下面的函数：

```
:echo items({'a':100,'b',200})
```

Vim会输出一个嵌套的列表，类似于[['a',100],['b',200]]。据我所知，Vimscript里的字典是无需的，所以不要期望你拿到的条目列表是有序的！

你可以用keys和values函数分别取出键值和值的列表。它们会按照你的预期来工作的——试试它们。

练习

- 阅读:help Dictionary里所有的内容，注意大写的“D”。
- 阅读:help get()。
- 阅读:help has_key()。
- 阅读:help items()。
- 阅读:help keys()。
- 阅读：help values()。

Toggling

在之前的章节里，我们讲了怎么在vim里设置选项。对于布尔型选项，我们可以用set someoption!来切换选项的值。当我们对这个命令设置一个快捷键的话，它就变的非常有用了。

运行下面的命令：

```
:nnoremap N :set local number!
```

在normal模式下输入N来试试上面的命令。Vim会对当前窗口的行号选项进行切换。创建一个这样的切换映射是非常有用的，因为我们没必要对于一个选项的开和关使用不同的两个按键。

不幸的是，这只对布尔型选项有效。如果你想切换一个非布尔型的选项的话，我们需要一些额外的工作。

切换选项

我们从创建一个用来切换选项的函数开始，然后建立一个映射来调用这个函数。把下面的代码放到你的~/.vimrc文件里（或者一个单独的文件放在~/.vim/plugin目录下）：

```
nnoremap  :call FoldColumnToggle()

function! FoldColumnToggle()
     echom &foldcolumn
endfunction
```

写入文件并且加载，然后通过来试试上面的映射。vim会输出当前的foldcolumn选项的值。如果你对foldcolumn这个选项的作用不是很明白的话，你可以先阅读一下:help foldcolumn。

下面我们加入实际的功能。编辑代码，修改成如下内容：

```
 nnoremap  :call FoldColumnToggle()

function! FoldColumnToggle()
     if &foldcolumn
          setlocal foldcolumn=0
     else
          setlocal foldcolumn=4
     endif
endfunction
```

写入文件，并加载。每次你输入的时候vim会显示或者隐藏折叠列。

这里的if语句只是检查foldcolumn的值是否为真值（在vim里整数0代表假，其他的数字代表真值）。所以，如果是假值的话，就会把它设置成0，这样就会隐藏它。否则的话就把它的值设置为4。很简单。

你可以用类似的简单函数来操作任何的0值代表关闭，其他值代表打开的选项。

切换其他功能

选项并不是我们想要进行切换的唯一功能。一个很有用的功能就是对快速补全窗口设置快捷键。我们先像之前一样先写一个简单的框架。把下面的代码加入到你的文件里：

```
nnoremap  :call QuickfixToggle()
```

```
function! QuickfixToggle()
     return
endfunction
```

这个映射并不会做任何事情。我们来把改变它，是它变得有用起来（不过并没有完全完成）。修改代码如下所示：

```
nnoremap  :call QuickfixToggle()

function! QuickfixToggle()
     copen
endfunction
```

写入并且加载这个文件。如果你再试试这个映射的话，你会看到它只是打开了一个快速补全窗口。

为了达到我们要切换的目的，我们先用一个简单，但是很不好的方法：全局变量。修改代码如下所示：

```
 nnoremap  :call QuickfixToggle()

function! QuickfixToggle()
        if g:quickfix_is_open:
               cclose
               let g:quick_fix_is_open = 0
        else
               copen
               let g:quick_fix_is_open = 1
         endif
endfunction
```

我们所做的很简单——我们用一个全局变量来表示快速补全窗口的状态，每次改变状态的时候，都把当前的状态存入到这个变量里。

写入，并且加载这个文件，试着运行这个映射。vim会抱怨说这个变量没有定义！我们来修复这个问题，对它进行一下初始化：

```
nnoremap  :call QuickfixToggle()

let g:quick_fix_is_open = 0

function! QuickfixToggle()
        if g:quickfix_is_open:
               cclose
               let g:quick_fix_is_open = 0
        else
               copen
               let g:quick_fix_is_open = 1
         endif
endfunction
```

写入并且加载这个文件，再试试这个映射，它起作用了。

提高

我们的切换功能起作用了，但是它还是存在一些问题。

第一个问题是，如果用户手动用:copen和:cclose来打开和关闭这个窗口的话，那么我们的变量的值就没法相应地进行更新了。这个问题并不是很严重，因为大部分情况下用户都会用映射来打开这个窗口的，如果不是的话，他们只需要多按一次映射键就可以了。

这个也说明了一个道理：当你在写vimscript代码的时候，如果你想要把所有的边界条件都覆盖到的话，你会陷入困境，并且很难完成你的任务的。

让一个功能大部分情况下都能正常工作（并不是说在不工作的时候爆发出一大堆问题）以及继续编码会比花大量的时间来让它达到100%的完美更好。不过一个例外就是，当你在写一个插件供很多人使用的时候。这个情况下，你最好多花点时间，让它能够处理绝大多数的场景，这样会让你用户开心，同时也会减少bug报告。

还原Windows/Buffers

我们的函数的另外一个缺陷是，如果用户使用映射的时候他已经在快速补全窗口中了，那么vim会关闭它，并且把它切换到上一个窗口里，而不是把它放到它之前所在的地方。这个在你想很快检查一下快速补全窗口里的选项并且继续工作时，比较让人讨厌。

为了解决这个问题，我们引入一个写vim插件会迟早会用到的功能。修改你的代码如下所示：

```
let g:quick_fix_is_open = 0

function! QuickfixToggle()
        if g:quickfix_is_open:
               cclose
               let g:quick_fix_is_open = 0
               execute g:quickfix_return_to_window . "window w"
        else
               let g:quickfix_return_to_window = winnr()
               copen
               let g:quick_fix_is_open = 1
         endif
endfunction
```

我们在函数的代码里添加了新的两行。第一个（在else语句里），会在运行:copen之前，把当前窗口的编号存入到另一个全局变量里。

第二行（在if语句里）执行wincmd，并且用窗口的编号作为前缀，来告诉vim跳转到对应的窗口。

再次说明一下，我们的方案并不是完美的。因为我们的用户可能在运行映射的时候打开或者关闭一个新的split。即使是这样，这个功能对于现在而言已经足够好了。

这种手动保存全局变量的方法在一些很严肃的程序里是不可取的，但是对于简单的vimscript函数而言这是一个快速但是恶心的可以让你的功能很快完成，并且继续你的生活的方式。。

练习

- 阅读:help foldcolumn
- 阅读:help winnr()
- 阅读:help ctrl-w_w
- 阅读:help wincmd。
- 阅读:help ctrl-q和:help ctrl-f来看看，你的这些映射覆盖了什么
- 如果需要的话通过s:和来加上命名空间

函数式编程

我们现在来花一点时间讨论一种你或许听过的编程方式：函数式编程。

如果你曾经使用过Python，Ruby或者Javascript，或者特别是Lisp，Schema，Clojure或者Haskell，那么你就会对对把函数当作变量使用以及使用不可变的变量比较熟悉了。如果你没有的话，你也可以安全地跳过这个章节，但是我还是推荐你学习本章来拓宽你的视野。

Vimscript里包含了函数式编程所需要的所有功能，不过它有点笨拙。我们首先来创建一些帮助函数，来使得函数式编程不那么困难。

创建一个functionnal.vim文件，这样你就没有必要来回地敲所有代码了。这个文件是我们本章的便签条。

不可变的数据结构

不幸的是，在vim里，没有任何内置的不可变数据结构类似Clojure的vector和map，不过我们可以通过创建一些帮助函数来在某种程度上来模拟不可变数据结构。

把下面的代码添加到你的文件里：

```
function! Sorted(1)
     let new_list = deepcopy(a:1)
     call sort(new_list)
     return new_list
endfunction
```

写入，并且加载这个文件，然后通过:echo Sorted([3,2,4,1])来测试它。Vim输出“[1,2,3,4]”。

那么这个包装函数和直接调用内置的sort()函数有什么区别呢？关键在于第一行代码：let new_list = deepcopy(a:1)。Vim的sort()函数对原来的list进行原地排序，所以我们要创建这个列表的一个拷贝，然后对这个拷贝进行排序，这样的话原始的列表就不会被改变了。

这样就可以避免任何副作用了，这样就可以帮助我们写出更容易维护和测试的代码了。下面我们添加一些类似的帮助函数：

```
function! Reversed(1)
     let new_list = deepcopy(a:1)
     call reverse(new_list)
     return new_list
endfunction

 function! Append(1,val)
     let new_list = deepcopy(a:1)
     call add(new_list,a:val)
     return new_list
endfunction

 function! Assoc(1,i,val)
     let new_list = deepcopy(a:1)
     let new_list[a:i] = a:val
     return new_list
endfunction

 function! Pop(1,i)
     let new_list = deepcopy(a:1)
     call remove(new_list,a:i)
     return new_list
endfunction
```

上面的函数基本上都是相同的，除了中间几行不同以及传入的参数不同。写入和加载上面的文件，然后通过几个列表来测试它们。

Reversed() 返回一个列表的反序

Append() 返回一个新的列表，它是在旧的列表末尾加上给定的元素

Assoc()（associta的简写）返回一个新的列表，它是把旧列表在指定位置上的值替换成给定的值

Pop() 返回一个新的列表，它是就列表删除了末尾的元素

函数当作变量

Vimscript支持用变量来引用函数，不过语法上有些怪异。运行下面的命令：

```
:let Myfunc = function("Append")
:echo Myfunc([1,2],3)
```

Vim会输出[1,2,3]。要注意的是我们使用的变量的名称是大写字母开头的。如果一个vimscript变量是引用一个函数的，那么它就必须以大写字母开头。

函数也可以像其他变量一样，可以保存在列表里。运行下面的命令：

```
:let funcs =[function("Append"),function("Pop")]
:echo funcs[1](['a','b','c'],1)
```

Vim会输出['a','c']。funcs这个变量没必要用大写字母开头，因为它保存的是一个列表，而不是函数，列表是什么内容对它没影响。

高阶函数

我们来创建一些常用的高阶函数。如果你对它们不熟悉的话，简单来说，高阶函数就是用其他函数做为参数，然后用它们来做一些操作的函数。

我们先从map函数开始。把下面的代码添加到你的文件里：

```
function! Mapped(fn,1)
     let new_list = deepcopy(a:1)
     call map(new_list,String(a:fn) . '(v:val)')
     return new_list
 endfunction
```

写入并且加载上面的文件，然后用下面的命令来试试上面的函数：

```
:let mylist = [[1,2],[3,4]]
:echo Mapped(function("Resversed",mylist)
```

Vim会输出[[2,1],[4,3]]，也即是对给定的列表里的每一个元素应用了Reversed函数。

那么Mapped()函数究竟是怎么生效的呢？这次我们也是用deepcopy来复制一个新的列表，然后对这个列表进行一些操作，然后返回这个修改后的列表——没有什么新的东西。但是真正的玄机就在中间一部分代码里。

Mapped()接受两个参数：一个函数的引用（也就是vim里指向一个函数的变量）以及一个列表。我们用内置的map函数来进行实际的操作。现在阅读一下:help map()来看看它到底是怎么工作的。

现在，我们再来创建一些其他常用的高阶函数。把下面的代码添加到你的文件里：

```
function! Filtered(fn,1)
     let new_list = deepcopy(a:1)
     call filter(new_list,String(a:fn) . '(v:val)')
     return new_list
endfunction
```

用下面的命令来试试Filtered函数的功能：

```
:let mylist = [[1,2],[],['foo'],[]]
:echo Filtered(function('len'),mylist)
```

Vim会输出[[1,2],['foo']]

Filtered接受一个描述性的函数和一个列表。它依次作用列表的元素，以它们为参数调用给定的函数，最后返回一个包含所有调用该函数的返回值为真值的元素的列表。在这里，我们用了内置函数len，所以会过滤掉所有长度为0的元素。

最后我们创建一个和Filtered功能相反的函数：

```
function! Removed(fn,1)
     let new_list = deepcopy(a:1)
     call filter(new_list,'!' . string(a:fn) . '(v:val)'
     return new_list
endfunction
```

使用上面的例子再试试这个函数：

```
:let mylist = [[1,2],[],['foo'],[]]
:echo Removed(function('len'),mylist)
```

Vim会输出[[],[]]。Removed和Filtered很相似，只不过它保留了描述函数返回为假的元素。

上面两段代码唯一的区别就在于下面的代码多了一个“!”。我们把它加在call命令里，用来对描叙函数的返回值进行取反。

性能

或许你会认为在上面的函数里，都进行了数组的拷贝，这样是很浪费的，这样会导致vim需要不断地创建新的数组，并且回收就的数组。

如果这样来考虑的话，你的想法是对的！Vim并不支持类似于Clojure向量的结构共享的功能，所以所有的拷贝操作都是要很大的开销的。

不过只是有些时候这个问题才会出现。如果你要操作相当多的数组，那么速度就会慢下来。不过，在实际生活中，你会为你基本上很少注意到这样的差别而感到惊讶。

考虑这样一个场景：当我在写这个章节的内容的时候，我的Vim值占用了大学80M的内存（同时我安装了很多的插件）。我的笔记本有8G的内存。那么几个列表的拷贝对我有实质性的影响么？当然，这个问题取决于列表的答谢，但是在绝大多数的场景下，是没有影响的！

相比而言，我的Firefox浏览器打开5个标签页就使用了1.22G的内存。

你需要自己来判断这种方式的代码是否会带来不可接受的性能问题。我极力赞同你进行一个反例的常识。使用不可变的数据

结构以及高阶函数可以简化你的代码，同时也会减少一大堆和状态相关的bug。

练习

- 阅读:help sort()
- 阅读:help reverse()
- 阅读:help copy()
- 阅读:help deepcopy()
- 阅读:help map()
- 阅读:help function
- 修改Assoc()，Pop(),Mapped()，Filtered()以及Removed()让它们能够支持字典。这里你会需要阅读:help type()
- 实现Reduced()
- 喝一杯你最喜欢的饮料，这个章节还是比较难的！

路径

Vim是一个文本编辑器，并且文本编辑器经常是对文本文件进行操作。文本文件是在文件系统里进行管理的，我们需要通过文件路径来指定文件。当你需要对路径进行操作的时候，vim提供了一些很有用的内置函数。

绝对路径

有些时候外部脚本取得某些文件的路径是很简单的。运行下面的命令：

```
:echom expand('%')
:echom expand('%:p')
:echom fnamemodify('foo.txt',':p')
```

第一条命令用来显示你当前在编辑的文件的相对路径。%代表“当前文件”。vim里还有很多可以用expand()来进行替换的字符串。

第二条命令会输出当前文件的全路径。字符串里的:p用来告诉vim取文件的全路径。vim里也有很多这样的修饰符可以使用。

第三条命令根据当前路径显示foo.txt文件的绝对路径，而不用考虑这个文件是否存在。fnamemodify()是一个vim函数，它比expand()函数更加灵活，因为它可以指定任何文件，不仅仅只是expand()的特殊字符。

列出文件

有时候你会需要取得某个目录下的文件列表。运行下面的命令：

```
:echo globpath('.','*')
```

Vim会列出当前目录下的所有文件。globpath()函数返回一个字符串，包含所有的文件名，文件名之间用回车分割。如果想要拿到第一个列表的话，你可以自己用split来切分。运行下面的命令：

```
:echo split(globpath('.','*'),'\n')
```

这次vim会输出一个包含所有文件的列表。

golbpath函数的通配符基本上也是按照你所认为的方式去工作的。运行下面的命令：

```
:echo split(globpath('.','*.txt'),'\n')
```

Vim输出当前目录的下所有的以.txt结尾的文件的列表。

你可以通过**来递归地列举当前文件夹下的所有文件。运行下面的命令：

```
:echo split(globpath('.','**'))
```

Vim会输出当前文件夹下的所有文件和文件夹。

globpath的功能非常强大，在你完成本章的练习之后，你会学到更多东西。

练习

- 阅读 :help expand().
- 阅读 :help fnamemodify().
- 阅读 :help filename-modifiers.
- 阅读 :help simplify().
- 阅读 :help resolve().
- Read :help globpath().
- Read :help wildcards.

创建一个完整的插件

前面的40个章节里，我们学习了很多基本的知识。在这本书最后的章节里，我们会从头开始为一门编程语言实现一个插件。

这个内容并不是为了让你畏惧，而是让你多做点努力。

如果你现在想停止的话，也是可以的！因为你现在已经学到了足够的内容里，你可以在你的vimrc文件里添加一些很强大的功能，也可以修复你在别人的插件里发现的bug。

如果你想说“现在学到的已经足够了——我不想浪费几个钟头的时间来创建我并不经常使用的插件”也没什么。实际点，如果你现在还不想做一个完整的插件的话，你现在就可以停止了，等你需要的时候可以从这里继续。

如果你想继续的话，那么你最好保证你会花一些时间的。接下来的内容会比较难，我会假设你真的要学习这些内容，而不是在你的沙发上浏览它。

Potion

我们要创建的插件是要对Portion编程语言进行支持。

Potion是一个玩具样的编程语言。它是一个很小的语言，这也使得它非常适合作为我们的目标。

Potion和Io非常相似，也有一些Ruby，Lua以及其他语言特点的混合。如果你之前没有接触过Io的话，你可能会感觉这个比较古怪。我强烈建议你先花一两个小时的时间来熟悉这门语言。你或许不会使用它，但是它也许会改变你思考的方式，同时会向你展示一些新的观点。

Potion解释器现在的实现版本里有些恶心的地方。例如，如果你的代码里有语法错误的话，解释器会出现段错误。所以不要在这个地方阻塞住——我会提供一些正确的代码，来保证你只需要关注Vimscript而不是Potion。

我们的目标不是学习Potion（尽管我认为学习它会让你成为一个更好的程序员）。我们的目标是把Potion作为一个例子，使得我们能够接触到编写Vim插件的方方面面的内容。

练习

- 下载和安装Potion。这个比较简单，需要要你自己完成。
- 确保小册子里的几个例子能够被Potion解释器正确执行。如果不行的话，到这里看看是什么原因导致的。

黑暗时代里插件的布局

我们要讨论的第一件事情是怎么样来结构化我们的插件。在过去，这是一个很繁琐的事情，但是现在有一个工具可以让插件的安装便得更加方便了。

首先我们要讨论一些基本的布局，然后再来看看怎么样使用方便的方法。

基本布局

一般情况下，Vim支持把插件拆分成多个文件。~/.vim目录下有多个文件夹来分别存放不同功能的文件。

不要紧张，我们只是讨论最重要的一些目录。在我们后面创建插件的时候，我们会一次讨论这些文件的。

在我们继续之前，我们需要先讨论一些词汇。

我一直把“插件”定义为“一大片做一连串相关事情的Vimscript代码”。Vim有对“插件”有一个更加明确的定义，即使~/.vim/plugins/目录下的一个文件。

大部分的时间我们会用第一个定义，如果有时候你觉得这个定义不清晰的话，那就请告诉我，我会试着来改写它。

~/.vim/colors

~/.vim/colors 目录下的文件都会被当作颜色方案。例如：如果你运行:color:mycolors命令，Vim会查找一个~/.vim/colors/mycolors.vim文件，并且执行它。这个文件需要包含所有的生成你的颜色方案的Vimscript命令。

我们这本书里不讨论颜色方案的设计。如果你想要设计自己的颜色方案的话，那么你应该拷贝一份现成的方案，并且修改它。记住:help是你的好朋友。

~/.vim/plugin

~/.vim/plugin目录下的文件会在每次vim启动的时候被运行的。这里的文件包含了那些你想要Vim在启动的时候加载的代码。

~/.vim/ftdetect

这个目录下的所有文件也都会在vim启动的时间加载运行。

“ftdetect”代表“filetype detect”。这个文件夹里的文件只需要设置自动命令来探测和设置文件的filetype。这也就表示它不能超过一行或者两行的长度。

~/.vim/ftplugin

~/.vim/ftplugin 目录下的文件都不同。

文件的名称很关键。当Vim把一个缓冲的filetype设置成一个值的时候，它会在~/.vim/ftplugin/目录下找到对应的文件。例如：如果你运行set filetype=derp，vim会查找~/.vim/ftplugin文件。如果这个文件存在的话，vim就会执行它。

Vim也支持~/.vim/ftplugin下存放目录文件。我们继续上面的例子：set filetype=derp，会让vim运行~/.vim/ftplugin/derp/目录下的所有*.vim文件。这样就可以让你把你的插件的ftplugin文件放在一个分组里。

因为这些文件在每次缓冲区的filetype改变时都会被执行，所以你只能设置缓存相关的本地变量。如果设置全局变量的话，会覆盖所有缓冲区的设置。

~/.vim/indent/

~/.vim/indent/下的文件和ftplugin文件很相似——它们都是根据文件名来加载的。

indent文件应该设置对应类型的文件的缩进选项，并且这些选项都是本地缓存相关的。

当然，你也可以把它放在ftplugin目录下，但是最好还是放在indent目录下，因为这样别的Vim用户就会很方便地理解你所做的事情。这个只是一个方便的用法，但是请做一个考虑周到的插件作者，并且遵循这个用法。

~/.vim/compiler/

~/.vim/compiler/目录下的文件和indent下的文件的工作方式完全一样。它们会根据自己的文件名来设置当前缓存的编译器相关的本地选项。

现在不必要考虑什么是编译器相关的选项，我们后面会讨论它的。

~/.vim/after

~/.vim/after/的作用有点黑。这个目录里的文件会在vim每次启动的时候进行加载，但是都是在~/.vim/plugin/下的文件加载完之后。

这样就允许你覆盖vim的内部文件的配置。实际上，你很少会用到它，不用担心这个文件夹，除非你想要改变vim的一些默认值。

~/.vim/autoload/

~/.vim/autoload/文件夹是一个非常有用的文件夹。它实际上的功能比它的名字听起来更加复杂。

简而言之：autoload是一个可以延迟加载你的插件的代码知道真正需要加载的一种方式。我们会在我们需要重构我们的插件的时候，再详细讨论这个，并且使用它。

~/.vim/doc/

最后，~/.vim/doc文件夹是你放插件的文档的地方。Vim在文档上花了很大的功夫（通过我们每次运行的:help命令可以证明它），所以给你的插件写好文档也是很重要的。

练习

- 重新阅读这个章节。我不是开玩笑。确保，你能够大致了解每个文件夹的下所包含的文件的作用。

新的希望：用**Pathogen**来管理插件布局

Vim最基本的插件布局可以让你在不同的地方放不同的文件来自定义你自己的Vim，但是当你使用别人的差距的时候就会感觉一团糟了。

在过去，当你想用其他人写的一个插件时，你需要下载所有的文件，并且你需要把它们一个一个地放到正确的文件夹里。你也可以用zip或者tar命令来帮你完成这个工作。

这种方法有几个很明显的问题：

- 如果你要更新一个插件怎么办呢？你可以覆盖旧的文件，但是你怎么知道作者删除了哪些文件，并且你需要手动去删除它。
- 如果两个插件有一个同名的文件怎么办呢（例如utils.vim或者其他常用的名字）？有些情况下你可以手动去编辑它，但是如果文件是在autoload或者其他的文件名比较关键的目录里的时候，那么你就需要自己修改插件的代码了。这样是很不好的。

  人们想了很多巧妙的方法来想使得上面的问题变得简单，例如Vimballs。幸运的是，我们再也不需要使用这些恶心的方法了。Tim Pope创建了一个很有用的插件，这个插件使得多插件的管理变得十分轻松，它只需要插件的作者把它的插件布局正常即可。

我们先来看看Pathogen是如何工作的，已经为了保持兼容我们该做些什么。

运行时路径

当vim查找一个特定目录（例如syntax/）下的文件时，它不只是查找一个地方。就像Linux/Unix/BSD系统有PATH变量一样，Vim有runtimepath选项来告诉它去哪儿寻找要加载的文件。

在你的桌面创建一个color文件夹，在这个目录里创建一个名叫mycolor.vim的文件（在这个例子里你可以保持这个文件为空即可）。打开vim运行下面的命令：

```
:color mycolor
```

Vim会报错，因为它不知道是要到你的桌面去寻找对应的文件。现在运行下面的命令：

```
:set runtimepath=/Users/sjl/Desktop
```

你需要修改上面的路径，让它对应到你的桌面。再试试上面的命令。

这次Vim不会保存了，因为它能够找得到mycolor.vim文件。因为这个文件是空的，所以它实际上什么都没有做，不过我们知道vim找到它了，因为Vim这次没有报错。

Pathogen

Pathogen插件会在打开Vim的时候添加路径到runtimepath变量里。所有在~/.vim/bundle/目录下的文件都会被加到runtimepath里。

也就是说，所有在bundle/目录下的文件夹，都需要包含部分或者所有的标准vim插件目录，例如colors/和snytax/。Vim然后就可以从这些文件夹里加载所需的文件了，这样就只需要把插件的文件统一放在一个目录下，操作起来更加简单了。

兼容**Pathogen**

当我们写Potion插件的时候，我们希望用户能够通过Pathogen来安装它。实现这个目的的方式很简单：我们只需要把文件放

到插件资源库下对应的目录里即可。

我们的插件资源库最终的结构如下所示：

```
potion/
     README
     LICENSE
     docs/
          potion.txt
     ftdetect/
          potion.vim
     ftplugin/
          potion.vim
     syntax/
          potion.vim
     ...etc...
```

我们可以把上面的内容放到GitHub或者Bitbucket上，这样的话，用户就可以直接把它拷贝到bundle/目录下，然后就可以直接使用了。

练习

- 安装Pathogen插件，如果你还没有安装的话。
- 为你的插件创建一个Mercurial或者Git仓库，命名为potion。你可以把它放在任何地方，然后建立一个符号链接到~/.vim/bundle/potion/或者干脆就放在~/.vim/bundle/potion/目录下。
- 创建README和LICENSE文件，并且提交到代码库里。
- 把仓库里的内容推送到Bitbucket或者GitHub上。
- 阅读:help runtimepath。

检测文件类型

我们先创建一个Potion文件用来实验我们的插件。创建一个factorial.pn文件，然后把下面的Potion代码写进去：

```
factorial = (n):
     total = 1
     n to 1 (i):
          total *= i.
     total.

10 times (i):
     i string print
     '! is: ' print
     factorial (i) string print
     "\n" print.
```

这段代码创建了一个简单的factorial函数，然后调用它10次，然后每次调用都输出对应的结果。用potion factorial.pn命令来运行它。输出的结果应该和下面相似：

```
0! is: 0
1! is: 1
2! is: 2
3! is: 6
4! is: 24
5! is: 120
6! is: 720
7! is: 5040
8! is: 40320
9! is: 362880
```

如果你没有看到这样的输出，或者是有错误的话，先找出问题所在。这段代码应该是正确的。

花点时间来理解一下这些代码是怎么工作的。你随时都可以参考参考Potion。这不仅仅是对理解Vimscript代码有用，并且可以让你成为一个更好的程序员。

检测文件类型

打开factorial.pn文件，运行下面的命令：

```
:set filetype?
```

Vim会输出filetype=，因为它现在还不知道.pn文件的类型。让我们来给它加上类型的判断。

在你的插件目录下添加ftdetect/potion.vim。添加下面的代码：

```
au BufNewFile,BufRead *.pn set filetype=potion
```

这里创建了一个简单的autocmd：这个命令设置.pn文件的类型为potion。还是比较简单的。

注意这里我们没有像之前一样用autocmmand group，这是因为vim会自动把ftdetect/*.vim文件的内容用autocommand group来进行包装，所以你不用担心这个了。

关闭factorial.pn文件，再次打开它。现在运行之前的命令：

```
:set filetype?
```

这次Vim会输出filetype=potion。当Vim启动的时候，它会加载~/.vim/bundle/potion.vim里的autocommand group，当打开factorial.pn文件的时候，这个命令就会执行，然后文件的filetype就被设置成potion。

现在，我们已经让Vim能够识别Potion文件的类型了，我们可以继续来给我们的插件添加一些更加有用的功能了。

练习

- 阅读:help ft。不用担心，如果你什么都不明白的话。
- 阅读:help setfiletype。
- 修改Potion插件的ftdetect/potion.vim文件，用setfiletype命令来代替set filetype。

基本语法高亮

现在我们已经搞定了基本的东西了，现在是时候来给我们的Potion插件写一些有用的代码了。我们先从一些简单的语法高亮开始。

在你的插件目录里创建syntax/potion.vim文件。把下面的代码放进去：

```
if exists("b:current_syntax")
     finish
endif

echom "Our syntax highlighting code will go here."

let b:current_syntax = "potion"
```

关闭Vim，然后用它打开你的factorial.pn文件。什么都没有发生，但是，如果你运行:messages，你会看到这个文件确实是被加载了的。

注意：当我让你打开Potion文件的时候，我的意思是让你在一个新的Vim窗口或者是实例里打开，而不是在一个新的split或者tab下打开。因为用Vim窗口打开的话，会让Vim重新加载所有和那个窗口有关的文件，但是用spilt打开的话就不会了。

第一行和最后一行是用来防止重复加载语法高亮配置的习惯用法。

高亮关键字

在后面的代码里我们会忽略文件开头的和结尾的if和let这样的模板。但是不要删除这些文本行，忘记它们就可以了。

把echom的代码换成下面的代码：

```
syantax keyword potionKeyword to times
highlight      link  potionKeyword Keyword
```

关闭factorial.pn文件，然后再打开它。to和time就会在你的颜色主题里被作为关键字进行高亮了。

上面的两行代码展示了Vim里基本的语法高亮的机构。为了高亮一片语法：

- 你首先需要定义一个语法块，这个可以使用syntax keyword或者其他相关的命令（这个我们后面会进行讨论的）来实现。
- 然后，你需要把语法块和高亮组进行关联。高亮显示组是你在颜色主题里定义的，例如“函数名应该是蓝色的”。

这样就是得插件的作者，按照他们自己的理解来定义语法块，然后来把它和常见的高亮显示组进行管理。它也允许主题创建者为一组编程语言的结构设计颜色，但是无须了解具体的编程语言。

Potion还有很多其他的关键词，我们还没有用到过，所以我们也要把它们加进来，进行高亮显示：

```
syntax keyword potionKeyword loop  times to while
syntax keyword potionKeyword if elsif else
syntax keyword potionKeyword class return

highlight   link  potionKeyword Keyword
```

首先，最后一行是不用变的。我们还是让vim把potionKeyword这个组里的所有文本当作Keyword来进行高亮显示。

现在我们有三行命令，都是以syntax keyword potionKeyword开始的。这个向我们展示了多次运行这个命令是不会重置语法组的——它会把它添加进去！这样你就可以逐个地添加语法组了。

具体怎么定义语法组完全取决于你自己：

- 你可以把所有东西都放在一行文本里完成
- 你也可以拆分成多行，保持每行文本不超过80哥字符，这样就便于阅读了
- 你可以对一个分组里的每一个元素单独定义一行
- 你也可以像我一样，把相关的元素放在一起进行定义

高亮函数

另外一个Vim的高亮显示组是Function。我们来给高亮脚本加上对Potion的一些内置函数的出力。修改syantax文件如下所示：

```
syntax keyword potionKeyword loop  times to while
 syntax keyword potionKeyword if elsif else
 syntax keyword potionKeyword class return

 syatax keyword potionFunction print join string

 highlight   link  potionKeyword Keyword
 highlight   link  potionFunction Function
```

关闭再打开factorial.pn文件，你会看到potion的内置函数都会被高亮显示的。

这个和关键词高亮的原理是一样的。我们定义了一个新的语法组，然后把它连接到一个不同的高亮组。

练习

- 思考一下，为什么第一行的if exists和最后一行的let命令非常有用呢。如果你不清楚的话，也不必担心。我也得找Tim Pope去确认一下。
- 浏览一下:help syn-keyword。多留意一下提到iskeyword的部分内容。
- 阅读:help iskeyword。
- 阅读:help goup-name，熟悉一下颜色主题的作者常用的集中高亮组。

高级语法高亮

到现在为止，我们已经给Potion文件定义了keyword和function这两个简单的语法高亮。

如果你没有做上一章的练习的话，你需要回到上一章去完成它们，因为这一节里我会假设你已经完成了上一节的练习。

事实上，你需要会过头把你所跳过的所有练习都补上。即使你认为它们是没有用的，但是为了高效地学习本书你需要完成他们。这个问题上请相信我。

高亮注释

一个很明显需要高亮的部分就是注释了。但是问题在于Potion的注视是以#开始的，而#绝大大多数情况下并不是在iskeyword里。

如果你不知道iskeyword代表什么的话，说明你没有听我的话。会过头去把那该死的练习给做了。我给每章加上练习并不是为了给你添加没用的外加作业，你确实需要完成它们来理解这本书的内容。

因为“#”不是一个关键词，所以我们需要用正则表达式来匹配它以及注释的内容。我们用syntax match代替syntax keyword来完成这个工作。把下面的内容加入到你的语法文件里：

```
syntax match potionComment "\v#.*$"
highlight link potionComment Comment
```

我不会再告诉你这些代码该放到哪个位置了，你是个程序员，你自己可以判断。

关闭再打开factorial.pn。在文件里随便哪个位置添加一些注释，你会发现它会被高亮显示。

第二行命令比较简单：它告诉Vim把potionComment这个语法组里的内容作为Comment来高亮。

第一行里有些新的内容。我们用syntax match来告诉vim去匹配正则表达式，而非原始的字符串。

注意我们的正则表达式是以\v开始的，这个是在告诉vim用“very magic”模式。如果你不是很确定它的作用的话，你可以重新阅读基本正则表达式那一章节。

在这个特别的列子里，“very magic”模式并不是必须的。但是在后面我们可能会改变这个正则表达式，然后会奇怪这个正则表达式为什么不正常工作了，所以我们推荐为了保持一致，使用“very magic”模式。

对于正则本身，它表示的意义比较简单：注释是由#开始的，包含所有的字符一直到当前行的结尾。

如果你需要一个完整的正则表达式的教程的话，你可以参考Zed Shaw的Learn Regex the Heard Way。

高亮操作符

Potion需要高亮的另外一部分是操作符。把下面的内容添加到你的语法文件里：

```
syntax match potionOperator "\v\*\="
syntax match potionOperator "\v/\="
syntax match potionOperator "\v\+\="
syntax match potionOperator "\v-\="
syntax match potionOperator "\v\*"
syntax match potionOperator "\v/"
syntax match potionOperator "\v\+"
syntax match potionOperator "\v-"
syntax match potionOperator "\v?"
```

关闭再打开factorial.pn文件。你会发现factorial函数中间的*=操作符现在被高亮显示了。

你发现的第一个现象应该是我把每个正则表达式作为单独的一行来显示，而不是像关键词一样进行分组。这是因为syntax match不支持在同一行包含多组的方式。

你应该也发现了，我在每个正则表达式的前面都加上了\v，即使它并不是必要的。这是因为我在写Vimscript的时候为了保持我的正则表达式的语法的一致性，即使这样会让我多输入几个字符。

你也许会奇怪我为什么不用“\v-\=?”这样一个正则表达式来匹配-和=。显然你可以那样做，它也会正常工作的。我只是把-和-=当作两个不同的操作符来使用，所以我把它们放在不同的行里定义。

把这些操作符分开来定义，可以简化正则表达式，但是会使得代码变得比较冗长。不过我喜欢这样，如果你有其他的想法的话，你自己可以决定该怎么去做。

并且我还没有把=定义成一个操作符。我们会在后面来完成它，因为我想把它留到后面的内容里。

因为我分别对-和-=用了不同的正则表达式，所以我必须要在-后面定义-=!

如果我使用了相反的顺序的话，并且在Potion文件里使用了-=，那么Vim就会匹配到-，并且高亮它，然后就会留下=没有被匹配了。这个说明了当你用syntax来组建语法组的话，每个分组只会处理文件里还没有被匹配的内容。

这个有点过度简化了，但是我现在还不想过多的陷入到细节里面。现在你只需要记住这一的一条经验规则就可以了：首先匹配大的语法块，再匹配小的。

现在我们已经学到这个内容了，那么就把=加入到操作符里：

```
syntax match potionOperator "\v\="
```

花点时间考虑一下你该在哪个位置放这些语法文件。如果你需要提示的话，重新阅读一下前面的章节。

练习

- 阅读:help syn-match
- 在我们的例子里，没有把：当作一个操作符。阅读Potion的文档，考虑一下什么时候需要把:当作一个操作符。如果你要把它当作操作符的话，那就加在语法文件里。
- 对于.和/做同样的工作。
- 添加一个Potion语法块来高亮数字，并且把它和Number高亮连接起来。注意Potion支持的数字有2，0xffaf，123.23，1e-2和1.9956e+2。记住平衡你处理边界情况的时间和边界情况被用到的次数。

更加高级的语法高亮

语法高亮是一个很广泛的主题，它的内容很容易就能写满一本书。

我们现在来讨论最后一个重要的内容，然后继续学习其他的内容。如果你想要学习更多的内容的话，你可以阅读:help syntax文档，同时你也可以参考其他人写的语法高亮文件。

高亮字符串

Potion，和其他编程语言一样，支持类似于“Hello,world”的字符串，我们需要把它们作为字符串来进行高亮。为了对字符串进行高亮显示，我们需要使用syntax region命令。把下面的命令添加到你的Potion语法文件里：

```
syntax region potionString start=/\v"/ skip=/\v\\./ end=/\v"/
highlight link potionString String
```

关闭再打开factorial.fn文件，你会开到文件末尾的字符串被高亮显示了！

最后一行代码的作用你应该很清楚了。如果你不明白的话，回头阅读以下前两章的内容。

第一行用区域来添加语法块。区域包含一个开始和一个结束，分别用来标识区域的开始和结束位置。Potion的字符串用引号开始，下一个引号表示字符串的结束。

其中的“skip”参数可以帮助我们跳过字符串中间的转移符，这样就可以处理类似于"She sai:\"Vimscript is tricky,but useful!\""。

如果不用skip参数的话，那么Vim匹配的文本就会在中间的"的位置就停止了，这并不是我们所期望的。

简而言之，skip的作用就是告诉Vim“一旦开始批评这个区域的文本，我希望你跳过任何匹配是行skip的文本，即使它在正常情况下被当作区域的结尾”。

花几分钟的时间来考虑一下这个问题。对于"foo \"bar"这样的字符串会发生什么事情呢？这个现象正常么？那个现象会一直是正常的表现么？关上这本书，花几分钟的时间好好考虑一下！

练习

- 给单引号的字符串添加上语法高亮。
- 阅读:help syn-region。
- 阅读上面的内容会比这章花的时间长很多的。去喝杯饮料，这是你该得的。

基本代码折叠

如果你从来都没有用过Vim的代码折叠功能的话，你就不会知道你错过了什么。阅读:help usr_28，然后在你的日常工作中花些时间来使用一下vim的代码折叠功能。如果你已经熟悉了它的使用的话，再回到本章的内容上来。

不同类型的代码折叠

Vim有6种不同的方式用来定义代码折叠

手动方式

你手动创建折叠，这些折叠会被保存在内存里，不过你每次关闭vim，在你下次继续编辑这个文件的这些折叠就会丢失。

这个方法适用于你需要用映射来方便地创建折叠的情景。不过我们这本书里不讨论这个内容，不过你需要记住这个，以免你碰到类似的需要使用这个方法的场景。

Marker

Vim可以根据你的文本里的特殊字符串来进行代码折叠。

大多数情况下，这些特殊字符串都是放在注释里的，例如（//{{{），但是在有些编程语言里，你可以利用语言自身的语法来实现，例如Javascript里的{和}。

不过在你的代码里加上纯粹只是为了然文本编辑器进行折叠的注释会让你的代码显得非常的难看，但是这个功能的有点在于你可以手动添加任意的代码折叠。这个功能在你想要把你一个比较大的文件按照一种特别的方式来进行折叠的时候会显得非常的有用。

Diff

这个特殊的代码折叠模式就在比较文件的时候使用的。这个我们不会花时间讨论的，因为这种折叠是由Vim自己进行处理的。

Expr

这种方式让你用Vimscript代码来定义折叠的文职。这个是最强大的一种方法，但是需要做的工作也是最多的，我们后面会讲解这种方法的。

Indent

Vim也可以使用你代码自身的缩进来觉得折叠的位置。相同缩进的文本行都会一起折叠，空行以及空格行都会简单的跟着附近的文本进行折叠。

Potion代码的折叠

我们再来看看我们的Potion示例代码：

```
factorial = (n) :
     total = 1
     n to 1 (i)
          total *= i
     total.

10 times (i) :
     i string print
     '! is : ' print
     factorial (i) string print
     "\n" print.
```

上面的代码中，函数以及循环的代码都是可以进行折叠的，这就意味着我们可以通过使用缩进来完成基本的代码折叠。

在我们开始之前，我们在total *= i这段代码的上方加上注释，这样我们就有一个多行的内部代码块了来进行测试了。后面你会在练习里发现我为什么要加这一行了，现在你只需要相信我就可以了。上面的文件现在应该和下面相似：

```
factorial = (n) :
     total = 1
     n to 1 (i)
          total *= i
     total.

10 times (i) :
     i string print
     '! is : ' print
     factorial (i) string print
     "\n" print.
```

在你的Potion插件的目录里新建一个ftplugin目录，然后在里面新建一个potion文件夹。最后，在potion文件夹里再建一个folding.vim文件。

记住，当一个缓冲区的filetype被设置成potion的时候，vim就会运行上面的folding.vim文件（因为这个文件在potion文件夹里）。

把所有代码折叠相关的代码都放在一个对应的文件里，这样的话可以很好的组织插件的不同功能的代码。

把下面的内容加入到这个文件里：

```
setlocal foldmethod=indent
```

关闭Vim，然后再打开factorial.pn文件。用zR，zM和za来试试代码折叠的功能。

一行代码就给我们带来了很好的代码折叠效果！这个确实很酷。

不过，你应该注意到了，factorial函数的代码块被折叠了，但是里面的内部循环部分的代码没有被折叠。究竟是怎么一回事呢？

事实上是因为当你是用indent来进行代码折叠的时候，Vim会忽略以#开始的文本行。这个在编辑C语言的时候很有效（在这里#是一个预编译符），但是在编辑其他的语言的时候就没有用了。

我们在ftplugin/potion/folding.vim里添加一个新的命令来修复这个问题：

```
setlocal foldmethod=indent
setlocal foldignore=
```

关闭然后再打开factorial.vim，现在这个内容的代码块也能够被正确的进行折叠了。

练习

- 阅读:help foldmethod；
- 阅读:help fold-manual；
- 阅读:help fold-marker和:help foldmarker；
- 阅读:hlep fold-indent；
- 阅读:help fdl和:help foldlevelstart；
- 阅读:help foldminlines；
- 阅读:help foldingnore。

高级代码折叠

在上一个章节里，我们用Vim的indent折叠来给我的Potion代码添加了一些方便的但是不完美的折叠方式。

打开factorial.pn文件，然后使用zM来折叠所有的代码。现在你的文件应该看起来和下面相似：

```
factorial = (n):
+---     5 lines: total = 1

10 times (i)
+--     4 lines: i  string print
```

打开第一个折叠区，文件看起来如下所示：

```
factorial = (n):
     total = 1
     n to 1 (i):
+--     2  lines: # Multiply the running total.
     total.

10 times (i):
+--     4 lines: i  string print
```

这样看起来也不错，不过我个人更加喜欢用一个代码块的第一行来折叠它所有的内容。这个章节里我们会写一些自定义的折叠代码，我们完成后，上面的代码就会折叠成下面的样子：

```
factorial = (n):
     total = 1
+---  3 lines: n to 1 (i):
     total.

+--     5 lines:10 times (i):
```

这样代码看起来就更加紧凑了，也就更加容易阅读了。如果你更加喜欢indent方法的折叠的话，也是可以的。不过，还是要学习本章的内容来实践一下Vim的折叠表达式。

折叠理论

知道vim是怎么来“考虑”折叠的，对于编写自定义代码折叠是很有作用的。下面是对这个的简单介绍：

文件的每一行文本都对应一个折叠等级。这个等级要么是0要么是一个正整数。

折叠等级为0的行不会被折叠

相邻的文本，如果等级相同的话，会被折叠在一起

如果一个等级为X的折叠是关闭的话，它后面所有的折叠等级大于或者等于X的文本行都会被折叠，直到遇到折叠等级大于X的文本行为止。

下面我们用一个简单的例子来说明这个。打开一个vim窗口，把下面的文本粘贴进去：

```
a
     b
     c
          d
          e
     f
g
```

通过下面的命令来打开indent折叠：

```
:setlocal foldmethod = indent
```

试试这个折叠方式，看看它是怎么进行折叠的。

现在通过下面的命令来看看第一行的折叠等级：

```
:echom foldlevel(1)
```

Vim会输出0。现在再看看第二行的折叠等级：

```
:echom foldlevel(2)
```

Vim会输出1.在看看第三行：

```
:echom foldlevel(3)
```

Vim会输出1。这就意味着第二和第三行都是等级为1的折叠。

下面显示了每一行的折叠等级：

```
 a              0
     b          1
     c          1
          d     2
          e     2
     f           1
g                0
```

再次阅读前面的内容。打开和关闭每个折叠，看看对应的折叠等级，确保你明白了折叠的原理。

一旦你对于每行文本的折叠等级对于创建不同的折叠的作用，继续阅读下一章节。

第一步：创建一个计划

在我们开始写代码之前，我们先对我们的折叠定义一些规则。

首先，缩进一致的文本要一起折叠，同时前一行文本要和它们放在一个折叠里，所以，类似于下面的文本：

```
hello = (name):
     'Hello, ' print
     name print.
```

会被折叠成：

```
+--     3 lines: hello = (name)
```

空行和它下面一行的等级保持一致，所以文本后面的空行都不会被包含进来。也就意味着下面的文本：

```
hello = (name):
     'Hello, ' print
     name print.

hello('Steve')
```

会被折叠成：

```
+--     3 lines: hello =():

hello('Steve')
```

而不是：

```
+--     4 lines: hello=()
hello('Steve')
```

这些规则都是个人的喜好，不过现在这些规则是我们要实现的折叠方式。

开始

在编写自定义折叠代码之前，我们先打开两个split。一个打开ftplugin/potion/folding.vim文件，一个打开我们的样本文件factorial.pn。

在之前的章节里，我们都是通过打开再关闭的方式来使得我们的folding.vim文件生效，但是实际上还有一种更加简单的方式。

记得之前讲过，无论什么一个缓存里的文件的filetype被设置成potion，所有在ftplugin/potion/里的文件都会被运行。这就意味着，你只需要在factorial.pn的split里运行:set ft=potion来让Vim重新加载折叠代码。

这样就比关闭再打开文件方便多了。唯一需要注意的就是你要保持folding.vim文件，否则没有保存的内容是不会生效的。

Expr Folding

我们接下来要用Vim的expr folding 来让我们的代码折叠变得更加灵活。

我们还需要把folding.vim里的foldignore去掉，因为它只会在indent folding下生效。同样的，我们还需要告诉vim去使用expr folding，所以我们把folding.vim的内容改变成如下所示：

```
setlocal foldmethod=expr
setlocal foldexpr=GetPotionFold(v:lnum)

function! GetPotionFold(lnum)
     return '0'
endfunction
```

第一行的代码是告诉vim使用expr folding。

第二行代码指定了Vim用来取得当前行的foldlevel的函数。当vim运行这段代码的时候，它会把想要知道foldlevel的文本的行号作为参数传给这个表达式。然后这个表达式会把用这个参数来调用一个自定义的函数。

最后，我们定义了一个简单的函数，它会对所有的文本行返回'0'。注意这里返回的是一个字符串，而不是一个整数，后面我们会讲到这个的。

接下来，保存folding.vim，然后通过在factorial.pn文件里运行:set ft=potion来重新加载折叠代码。这次，Vim不会做任何折叠，因为我们的函数对于所有的文本行都返回'0'。

空行

我们首先来看看空行这个特殊的文本。修改GetPotionFold函数，使得它看起来如下所示：

```
function! GetPotionFold(lnum)
     if  getline(a:lnum) =~? '\v^\s*$'
          return '-1'
     endif

     return '0'
endfunction
```

我们在代码里添加了一个if语句来处理空行，它是怎么工作的呢？

首先，我们使用getline(a:lnum)来去等当前行的内容。

我们把取得的内容和正则'\v^\s*$'进行匹配。记住\v会打开'very magic'模式。这个正则会匹配“从文本行开始到结束全部都是空格”的文本。

这里的比较用的是大小写敏感的比较符=~?。从技术的角度来讲，在批评空格的时候是没必要注意大小写的，但是我比较喜欢在比较字符串的时候显示写清楚。如果你喜欢的话，也可以用~=代替。

如果你需要复习Vim里正则表达式的使用的话，你可以重新阅读“基本正则表达式”那一章节，以及“Grep 操作符”的章节。

如果当前行包含非空格字符的话，就不会匹配成功，这样就返回‘0’。

如果当前行匹配成功的话（例如，它是一个空行或者只包含空格），我们就会返回字符串‘-1’。

前面我们讲过，一个文本行的foldlevel可以是0或者正整数，那么这里又是怎样的情况呢？

特殊的折叠等级

你自定义的折叠表达式可以直接返回文本行的折叠等级，也可以返回一个特殊的字符串来告诉vim来特殊处理。

‘-1’就是其中的一个特殊字符串。它告诉Vim当前行的折叠等级是未定义的。Vim会这样解释它“这一行文本的折叠等级和它前面一行以及后面一行中，折叠等级较低的一个保持一致”。

Vim可以把这些未定义的文本行链接起来，如果你有两个未定义折叠等级的文本行在一个等级为1的文本行前面的话，那么vim就会设置最后一个未定义的文本行的等级为1，然后倒数第二个未定义的文本行的等级为1，然后第一个为1.

区块移动理论

如果你从来都没有使用过Vim的区块移动命令（`[[`，`]]`，`[]`，`][`），先花一点时间阅读一下说明文档。现在就开始阅读`:help section`的内容。

是不是有点迷惑了？这是正常的，我第一次读这些内容也是这样的感受。我们将要通过写代码的方式来迂回地学习这些移动是怎么运作的，然后，在下一节里让我们的Potion插件来支持它们。

Nroff 文件

“区块移动”的四个命令概念上都是用来在文件的“区块”上来进行移动的。Nroff是一个和LaTex或者MarkDown相似的语言——它是用来编写那些以后需要来格式化的文本的（其实这就是UNIX man page 所使用的格式）。

Nroff 文件使用一些特定的“宏”来定义“区块头”。例如，下面是一段取自`awk` man page的内容：

```
.SH NAME                                                ***
awk \- pattern-directed scanning and processing language
.SH SYNOPSIS                                            ***
.B awk
[
.BI \-F
.I fs
]
[
.BI \-v
.I var=value
]
[
.I 'prog'
|
.BI \-f
.I progfile
]
[
.I file ...
]
.SH DESCRIPTION                                         ***
.I Awk
scans each input
.I file
for lines that match ...
```

这些以`.SH`开头的都是区块头，我在上面通过`***`把它们给标记出来了。那四个区块移动的命令都是用来在这些区块头直接移动光标的。

Vim会把所有以`.`以及所有的nroff区块头的“宏”开始的文本行作为区块头进行处理，哪怕你编辑的不是一个nroff文件。

你可以通过修改`sections`设置的方式来修改这些宏，但是Vim还是需要你在文本行的开头通过点号来开始，并且这些宏都要是成对的字符，这样的话，这些设置就不会给Portion文件带来过多的灵活性。

花括号

区块移动命令也会查找另外一个东西：一个开或者关的花括号(`{`或者`}`)作为一行的开头。

`[[`和`]]`查找左花括号，`[]`和`][`查找右花括号。

这个额外的小功能可以让你在类似C语言一样的文本区块里进行移动。而且不管你在编辑什么文件，这些规则都是通用的。

把下面的文本放进缓存里：

```
Test          A B
```

```
Test

.SH Hello      A B

Test

{              A
Test
}                B

Test

.H World       A B

Test
Test           A B
```

现在运行 `:set filetype=basic` 来告诉Vim这是一个BASIC文本，然后再来试试区块移动的命令。

`[[` 以及 `]]` 命令会在标记了 `A` 的文本行间移动，而 `[]` 和 `][` 命令则会在标记了 `B` 的文本行间移动。

这个向我们展示了Vim会一直使用这两条相同的规则来进行区块移动，即使是在那些这两条规则都不适用的文本里（例如BASIC）！

练习

- 再次阅读 `:help section`，现在你应该了解了区块移动了。
- 可以阅读 `:help sections` 来娱乐一下。

Potionq区块移动

既然我们知道了区块移动是如何工作的，那我们现在就可以把这些命令映射成可以在Portion文件里工作的方式了。

首先我们需要确定Potion文件里的“区块”是什么。因为我们有两组可以使用的区块移动命令，所以我们可以使用两组“模式”，这样我们的用户就可以选择使用他们自己喜欢的。

我们使用下面两种模式来定义Portion区块的开始：

- 1. 所有后面跟着一个空行并且是以非空字符开始的行，或者是文件的第一行。

- 1. 所有第一个字符非空，并且行内包含等号，且该行以冒号结尾。

现在使用简单扩展后的 `factorial.pn` 的文件，下面是我们的规则中会作为区块头的样子：

```
```

我们的第一条规则定义的比较自由。它直接把一个“顶级的文本块”定义成一个区块。

第二个定义就比较严格一点了。它把一个区块定义为一个函数定义。

自定义映射

在插件的代码库里创建一个 `ftplugin/option/sections.vim` 文件。这里我们将会放入区块移动相关的代码。记住，只要缓冲区的 `filetype` 是 `potion`，这里的代码就会执行。

我们要重新映射全部的4个命令，所以我们先创建一个大致的“框架性”文件：

```
noremap
```

外部命令

Vim遵循UNIX的“做好一件事”的哲学。Vim没有把所有的功能都集成到编辑器内部，而是通过把某些合适的功能代理到外部的命令中。

我们添加一些能够和Potion编译器进行互动的插件来让我们能够更好地进行互动。

编译

首先，我们需要添加一个命令来编译和运行当前的Potion文件。实现这个功能有很多种方式，但是我们这里只是简单地使用一个外部命令。

在你的插件的代码库里创建一个 `potion/ftplugin/potion/running.vm` 文件。我们会在这个文件里创建用来编译和运行Potion文件的映射。

```
if !exists("g:potion_command")
    let g:potion_command = "potion"
endif

function! PotionCompileAndRunFile()
    silent !clear
    execute "!" . g:potion_command . " " . bufname("%")
endfunction

nnoremap  r :call PotionCompileAndRunFile()
```

第一部分代码创建了一个全局变量来保存用来执行Potion的命令，当然是在这个变量还没有设置的情况下。我们之前有过类似的代码。

这种方式可以允许用户重新设置 `potion` 命令的路劲，如果 `potion` 不是在默认的 `$PATH` 路径下，用户可以在 `~/.vimrc` 文件里添加上`g:option_command = "/Users/sjl/src/potion/poiton"来进行覆盖。

最后一行添加了一个缓存本地的映射，它会调用我们上面定义的函数。注意，因为这个文件是房子 `ftdetect/potion` 目录下，所以每当文件的 `filetype` 被设置成 `potion` 的时候，这个脚本文件都会被执行。

实际上的功能实现是在 `PotionCompileAndRunFile()` 函数里。保存这个文件，打开 `factorial.pn` 文件然后按下 `<localleader>r` 来运行这个映射，看看会发生什么。

如果 `potion` 是在你的 `$PATH` 路径下，这个文件会运行并且你会在终端上看到它的输出结果（如果你使用的时GUI Vim，结果会在窗口的底部出现）。如果你看到一个 `potion` 命令找不到的错误，那么你需要在 `~/.vimrc` 文件里设置 `g:option_command` 的值。

我们来看看 `PotionCompileAndRunFile()` 是怎么工作的。

!命令

`:!` 命令（读作“帮”）可以在Vim里运行外部命令，并且输出它们的结果。通过下面的命令来试试这它：

```
:!ls
```

Vim会输出这个 `ls` 命令的结果，并且会有一个“按回车键或者输入命令继续”的提示。

通过这种方式运行命令，Vim并不会给命令提供任何输入。通过下面的命令来确认这个：

```
:!cat
```

输入几行文本，你会看到`cat`命令会把这些文本再回显回来，就像你在Vim外部运行命令一下。使用`Ctrl-D`来结束这个命令。

要想运行外部命令时没有`Press ENTER or type command to continue`这样的提示的花，使用`:silent !`运行下面的命令：

```
:slient !echo Hello, world.
```

如果使用的时MacVim或者gVim这样的图形界面的Vim，你就不会看见`Hello, world.`这样的输出了。

如果你运行的是终端下的Vim，输出的结果以来于你的配置。在运行完单独的`:silent !`命令后你需要运行`:redraw!`来恢复你的屏幕。

注意这个命令式`:silent !`而不是`:silent!`（注意空格？）！这两个是不同的命令，并且我们需要的时前面一个！Vim是不是很强大？

我们现在回头看看`PotionCompileAndRun()`函数：

```
function! PotionCompileAndRunFile()
    silent !clear
    execute "!" . g:potion_command . " " . bufname("%")
endfunction
```

首先，我们运行一下`silent !clear`命令，这样就会清空屏幕，并且不会显示`Press ETER...`的提示信息。这样就会保证我们只会看到当前运行程序的输出，这对于不停地跑同一个命令是很有帮助的。

接下来的一行我们用之前的`execute`来动态构建一个命令。它构建的命令和下面有点类似：

```
!potion factorial.pn
```

注意这里没有`silent`，所以用户需要按回车键回到Vim。这使我们的映射所需要的，所以我们的工作完成了。

展示字节码

Potion编译器有个允许你查看生产的字节码的选项。当你需要调试底层代码的时候，这个功能是很有用的。试试在shell里运行下面的命令：

```
potion -c -V factorial.pn
```

你会看到很多类似下面的输出：

```
-- parsed --
code ...
-- compiled --
; function definition: 0x109d6e9c8 ; 108 bytes
; () 3 registers
.local factorial ; 0
.local print_line ; 1
.local print_factorial ; 2
...
[ 2] move     1 0
```

```
[ 3] loadk    0 0   ; string
[ 4] bind     0 1
[ 5] loadpn   2 0   ; nil
[ 6] call     0 2
...
```

我们来添加一个映射，让用户能够在一个Vim窗口里查看当前Potion文件的字节码输出，所以用户能够浏览并且查看它们。

首先，把下面的一行代码添加到`ftplugin/potion/running.vm`的末尾：

```
nnoremap  b :call PotionShowBytecode()
```

这里没有什么特别的 —— 只是一个简单地映射，下面我们简单描述一下我们要实现的函数：

```
function! PotionShowBytecode()
    " Get the bytecode.

    " Open a new split and set it up.

    " Insert the bytecode.

endfunction
```

现在我们有了大致的框架，我们再来讨论如何来实现它。

system()

我们有很多方式可以实现这个功能，所以我们会选择一个对你而言很实用的方式。

运行下面的命令：

```
:echo system("ls")
```

你会在屏幕的下方看到`ls`命令的输出。如果你运行`:message`你也在那儿看到有输出。所以Vim里的`system()`函数会执行一个命令，并且把命令的

你可以传入第二个字符串作为`system()`的参数。运行下面的命令：

```
:echom system("wc -c","abcdefg")
```

Vim会输出7（有些填充的空格）。如果你通过上面的方式传入第二个参数，Vim会把它写入到一个临时文件，并且通过管道把它传入到命令的标准输入上。

回到我们的函数，编辑PotionShowBytecode()用下面的代码来完成框架的第一部分：

```
function! PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%"))
    echom bytecode

    " Open a new split and set it up.

    " Insert the bytecode.

endfunction
```

保存文件，在factorial.pn里运行:set ft=potion来重新加载它，然后使用<localleader>b映射。Vim会在屏幕的底部输出字节码，一旦你确定这段代

草稿分屏

接下来我们要打开一个新的分屏来展示结果。这样的话，用户就能够使用强大的Vim来浏览字节码，而不是在屏幕底部看一次就够了。

要完成这个功能，我们需要创建一个“草稿”分屏：这种分屏包含一个缓存区，但是这个缓冲区的数据不会被保存下来，并且每次运行映射的时候都会被覆

```
function! PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%"))

    " Open a new split and set it up.
    vsplit __Potion_Bytecode__
    normal! ggdG
    setlocal filetype=potionbytecode
    setlocal buftype=nofile

    " Insert the bytecode.

endfunction
```

这个新的命令很容易理解。

vsplit创建一个垂直的分屏，它的缓冲区名称叫做__Potion_Bytecode__。这个缓冲区的名称前后都加了下划线用来表示这个文件不是一个常规的文件（

接下来我们通过normal! ggG命令来删除这个缓冲区里的所有内容。第一次运行映射的时候这个命令不会做什么，但是接下来的时间里，我们需要重复利用

接下来我们需要为这个缓冲区做两个设置。首先，我们把这个缓冲区的文件类型设置为potionbytecode，表面它里面的内容。其次我们把buftype设置成n

接下来，我们把保存到bytecode变量里的字节码放到这个缓存里。接下来完成我们的函数：

```
function! PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%") . " 2>&1")

    " Open a new split and set it up.
    vsplit __Potion_Bytecode__
    normal! ggdG
    setlocal filetype=potionbytecode
    setlocal buftype=nofile

    " Insert the bytecode.
    call append(0, split(bytecode, '\v\n'))
endfunction
```

这个append()Vim函数接收两个参数：一个表示从哪一行之后开始添加文本，然后是一个列表的文本行。举个例子，试试下面的命令：

```
:call append(3, ["foo", "bar"])
```

这个命令会在当前缓冲区的第三行的下面添加两行内容foo和bar。这种情况下，会在第0行下面添加文件，也就是“文件的顶部”。

我们需要一个字符串列表来进行添加，但是使用system()函数我们拿到的时一个里面有换行符的单个字符串。我们使用Vim的split()函数来把这一长串文

现在我们已经完成了这个函数，我们可以来试试这个映射的功能了。当你在factorial.pn的缓冲区里运行<localleader>b命令的时候，Vim会打开一个新

练习

- 阅读:help bufname的内容。
- 阅读:help buftype的内容。
- 阅读:help append()的内容。
- 阅读:help split()的内容。
- 阅读:help :!的内容。
- 阅读:help :read和:help :read!的内容（虽然我们没有讲这些命令，但是它们是很有用的）。
- 阅读:help system()的内容。
- 阅读:help design-not的内容。

现在，我们的映射需要用字自己在运行映射之前先保存文件，这样他们的修改才会有效。撤销的动作很容易实现，所以我们修改函数来帮他们保存当前的文

当你在一个有语法错误的文件上来运行字节码映射的函数，会发生什么呢？为什么会是这样的结果呢？

修改PotionShowBytecode()函数，来检测当Potion编译器返回错误呢，把错误信息提示给用户。

额外加分

每次你运行字节码映射时，Vim会创建一个新的垂直分屏，即使用户没有关闭前一个。如果用户没有关闭它们，那么会出现很多窗口堆叠的情况。

修改PotionShowBytecode()函数，来检测是否已经有窗口为__Potion_Bytecode__缓冲区打开，如果有的话，直接切换到已有的窗口，而不是创建一个新

对于这个任务，你可能需要阅读以下:help bufwinnr()的内容。

更多额外的加分

记得我们把临时缓冲区的filetype设置成了potionbytecode吗？创建一个syntax/potionbytecode.vim文件，然后定义Potion字节码的语法高亮，使得

自动加载

现在我们已经给我们的Potion插件添加了足够多的功能，这也是我们在本书中要做得事情。在完成本书之前，我们会讨论一些能够让它更加闪亮的一些重要的方式。

第一件事是通过自动加载来让我们的插件更加有效率。

自动加载如何工作

现在，如果一个用户加载我们的插件（通过打开一个Potion文件）它所有的功能都一次性加载完成。我们的插件还比较小，所以没有什么问题，但是对于一个大型插件而言，加载所有的代码会消耗一定的时间的。

Vim的解决方案是通过“自动加载”的方式。自动加载让你可以延迟代码到实际用的时候才加载。这样会导致性能有一点点损耗，但是如果用户不是每次都是使用你插件的全部代码，自动加载会带来很好的提升。

下面我们来看看它是怎么运作的。先看看下面的命令：

```
:call somefile#Hello()
```

当你运行这个命令时，Vim的表现会和运行一个正常的函数有一些差别。

如果这个函数已经被加载了，Vim会直接调用它。

否则的花，Vim会在`~/.vim`目录（以及所有的Pathogen插件目录）查找一个`autoload/somefile.vim`文件。

如果这个文件存在，Vim会加载这个文件，然后再按照正常的方式来调用他们。

在这个文件里，函数的定义应该和下面的相似：

```
function somefile#Hello()
    " ...
endfunction
```

你可以在函数名里添加多个`#`来表示多个字目录。例如：

```
:call myplugin#somefile#Hello()
```

这样就会查找自动加载的文件`autoload/myplugin/somefile.vm`。并且文件内部的函数的定义需要带上全路径：

```
function myplugin#somefile#Hello()
    " ...
endfunction
```

体验

为了能够体验它是怎么工作的，我们先试试。创建一个`~/.vim/autoload/example.vim`文件然后添加如下内容：

```
echom "Loading..."

function! example#Hello()
```

```
    echom "Hello, world!"
endfunction

echom "Done loading."
```

保存文件，然后运行 `:call example#Hello()` 命令。Vim会输出如下内容：

```
Loading...
Done loading.
Hello, world!
```

这个小示例证实了以下几件事情：

- 1. Vim确实是在调用的时候才加载文件的。哪怕在我们打开Vim的时候文件不存在，所以它不可能是在启动的时候进行加载的！

- 1. 当Vim找到要加载的文件，它会在调用函数之前先加载文件。

不用关闭Vim，把函数的定义改成如下所示：

```
echom "Loading..."

function! example#Hello()
    echom "Hello AGAIN, world!"
endfunction

echom "Done loading."
```

保存文件，不用关闭Vim，运行 `:call example#Hello()` 函数。Vim会简单地输出：

```
Hello, world!
```

Vim已经有了对于 `exmpale#Hello` 函数的定义，所以它并不需要重新加载这个文件，也就是说：

- 1. 函数定义外面的代码没有再次执行。

- 1. 函数定义的改变并没有被重新加载。

现在运行 `:call example#BadFunction()` 。你会再次看到加载的信息，同时也会看到函数不存在的错误提示。现在再次运行 `:call example#Hello()` 。这次你会看到更新后的信息~

现在，你应该对于Vim如何加载自动加载形式的函数名的函数了：

- 1. 检查对应函数名的函数是否存在。如果存在，直接调用。

- 1. 否则，按照文件名加载文件，并执行执行文件。

- 1. 然后，调用函数。如果正常完成，就结束了。如果调用失败，就返回错误信息。

如果你还没有彻底理解上面的内容，回头再看看上面的示例内容，然后看看每条规则都在哪儿生效。

加载哪些内容

自动加载并不是完全免费得。除了代码里丑陋的命名规则外，对于加载而言还是有一定（小）的影响的。

我们已经说过，如果你创建一个不是每次用户打开Vim会话的时候都进行加载的差距，那么你就需要尽量把功能都移到延迟加载里面。这样会减少你的插件对于Vim启动时间的影响，尤其是在用户安装了很多插件的时候。

那么，哪些是可以被安全地自动加载的呢？答案是所有的那些不是被用户直接调用的代码。映射和自定义命令不能被自动加载（因为那样的话，用户就没法调用它们了），但是其他的很多都可以。

现在我们来看看我们的Potion插件，看看哪些是可以自动加载的。

给**Potion**插件添加自动加载

我们会从编译和运行的功能着手。回忆一下上一章节的末尾，我们的`ftplugin/potion/running.vim`文件的内容：

```
if !exists("g:potion_command")
    let g:potion_command = "/Users/sjl/src/potion/potion"
endif

function! PotionCompileAndRunFile()
    silent !clear
    execute "!" . g:potion_command . " " . bufname("%")
endfunction

function! PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%"))

    " Open a new split and set it up.
    vsplit __Potion_Bytecode__
    normal! ggdG
    setlocal filetype=potionbytecode
    setlocal buftype=nofile

    " Insert the bytecode.
    call append(0, split(bytecode, '\v\n'))
endfunction

nnoremap  r :call PotionCompileAndRunFile()
nnoremap  b :call PotionShowBytecode()
```

这个文件只会在Potion文件加载的时候调用，所以它不会对Vim的启动有影响。但是也有些用户不需要使用这些功能，所以如果可以使用自动加载，那么每次打开Potion文件的时候可以帮他们省几百毫秒。

当然，在这个场景下它不能节省多少时间。但是我可以肯定的时，假如一个插件有成千上万行函数的话，那么加载它们的时间也是很客观的。

我们开始吧。在你的插件代码库里创建一个`autoload/potion/running.vim`文件。然后把下面的两个函数移到这个文件里面，并且调整它们的名称，最终看起来如下所示：

```
echom "Autoloading..."

function! potion#running#PotionCompileAndRunFile()
    silent !clear
    execute "!" . g:potion_command . " " . bufname("%")
endfunction

function! potion#running#PotionShowBytecode()
    " Get the bytecode.
    let bytecode = system(g:potion_command . " -c -V " . bufname("%"))

    " Open a new split and set it up.
```

```
    vsplit __Potion_Bytecode__
    normal! ggdG
    setlocal filetype=potionbytecode
    setlocal buftype=nofile

    " Insert the bytecode.
    call append(0, split(bytecode, '\v\n'))
endfunction
```

注意 `potion#running` 是如何匹配它所在的目录和文件名的。现在修改 `ftplugin/potion/running`，内容如下所示：

```
f !exists("g:potion_command")
    let g:potion_command = "/Users/sjl/src/potion/potion"
endif

nnoremap  r
            \ :call potion#running#PotionCompileAndRunFile()

nnoremap  b
            \ :call potion#running#PotionShowBytecode()p
```

保存文件，关闭Vim然后打开你的 `factorial.pn` 文件。试试我们创建的映射，确保它们能够正常使用。

确保你只会在第一次使用映射的时候才会看到调试信息 `Autoloading...`（也许你需要使用 `:messages` 命令来查看它）。当你确定自动加载已经能够正常运行的时候，你就可以删除这些调试信息了。

正如你所看到的，我们现在只留下了 `nnoremap` 的映射。我们不能自动加载这些，因为如果这样做了用户就没有办法能够去自动加载这些映射了。

这个在Vim插件里是很常用的一种模式：大部分的功能都是自动加载的，只会在Vim每次都加载的文件里放上 `nnoremap` 和 `command` 命令。注意，当你写一个不小的插件的时候都要注意这样去做。

练习

- 阅读 `:help autoload` 的内容
- 继续试验一下，看看autoloading的变量是如何表现的。

加载你想编程实现如何加载一个Vim已经加载过的文件，并且不会打扰用户。也许你要试试这样做？你或许可以阅读一下 `:help silent!`。但是在现实中不要这么做。

文档

我们的的Potion插件有一些很有用的功能，但是如果我们不提供文档的话告诉他们这个插件能干什么的话，那么它对于任务人都是没有用的。

Vim自身的文档非常完美。它不啰嗦，但是覆盖了全部的功能。它的文档也知道了很多插件的作者来完善它们自己的文档，从而也就在Vimscript社区里形成了良好的文档文化。

文档是如何工作的

当你在Vim里阅读一个 `:help` 的话题时，你会发现相比较其他内容而言，有些内容是高亮的。我们来看看这个是如何做到的。

打开任何一个帮助文档（例如 `:help help` ），然后运行 `:set filetype?` 。Vim会显示 `filetype=hep` 。现在运行 `:set filetype=text` ，你会发现高亮的地方都消失了。再次运行 `:set filetype=help` 这些高亮又出现了。

事实上，Vim帮助文件和其他文件一样，都是语法高亮后的文本！这也就意味着你可以编写你自己的文档，然后也能够进行高亮。

在你的创建库里创建一个 `doc/potion.txt` 文件。当索引帮助文档的时候，Vim/Pathogen会查找 `doc` 文件夹下地内容，所以我们只需要在这里编写我们的帮助文档即可。

在Vim里打开这个文件，并且运行 `:set filetype=help` ，当你输入内容的时候，你会发现有语法高亮的功能。

帮助文件头

帮助文件的格式取决于个人的喜好。这样说了，那我就采取一种普遍被现代Vimscript社区所采用的一种比较流行的方式来组织它们。

帮助文件的第一行应该包含帮助文件的名称，接着是一行描述这个插件的文本。接下来就是你的 `potion.txt` 的第一行。

```
*potion.txt* functionality for the potion programming language
```

通过星号围起来的文本创建了一个可以被跳转的“标记”。运行 `:HelpTags` 命令来告诉Pathogen来重新构建帮助标签的索取，然后打开一个新的Vim窗口，运行 `:help potion.txt` 。Vim会像打开其他帮助文档一样打开你的帮助文档。

接下来，你需要把你的插件名称放在一长串的描述后面。有些作者（包括我）喜欢使用ASCII艺术字，来使得它变得更加有趣。添加一个不错的标题到 `potion.txt` 文件：

```
*potion.txt* functionality for the potion programming language

                      ___      _   _                 ~
                     / _ \___ | |_(_) ___  _ __      ~
                    / /_)/ _ \| __| |/ _ \| '_ \     ~
                   / ___/ (_) | |_| | (_) | | | |    ~
                   \/    \___/ \__|_|\___/|_| |_|    ~

          Functionality for the Potion programming language.
        Includes syntax highlighting, code folding, and more!
```

我是通过`figlet -f ogre "Potion"

练习

给Potion插件写文档。

阅读 `help help-writing` 了解一些关于写文档的帮助。

阅读 `:help :left`， `:help :right`，以及 `:help :center` 来学习有关如何使用这三个命令来让你的ASCII字符排版的更有艺术感。

我是通过运行 `figlet -f ogre "Potion"` 来得到这些有趣的字符的。Figlet是一个很有用的用来生成ASCII艺术字的小工具。每行末尾的 `~` 字符是用来保证Vim不高亮或者隐藏艺术字内部的字符。

文档写哪些内容

接下来一半都是目录。首先，我们来决定哪些需要写到文档里。

当给一个新插件写文档时，我一半都是按照下面的章节列表来编写：

- 介绍
- 用法
- 映射
- 配置
- 许可证
- 缺陷
- 如何贡献
- 更新日志
- 贡献列表

如果插件比较大并且需要一个“概述”，那么我会写一个介绍的章节来概要说明一下插件的原理。否则的话，我会直接略过这部分。

一个“用法”章节需要介绍一下用户实际上如何使用你的插件。如果他们是通过映射来操作的，就告诉他们这个。如果映射不多的花，你可以直接列出来，否则的话，你就需要一个单独的“映射”章节来把它们给列出来。

“配置”章节应该列出所有的用户可以修改的配置项，并且告诉用户它们的作用以及默认值。

“许可证”章节应该告诉用户插件的代码所使用的许可证，并且有一个链接指向许可证的内容。不要再你的帮助文档里放许可证的全部内容 —— 大部分用户都知道常见的许可证，放在里面只会显得比较混乱。

“缺陷”章节应该简单明了。列出所有的你已经知道的并且还没有修复的缺陷，并且告诉用户如他们发现了问题，该如何告知你。

如果你想让用户能够提供修复缺陷的代码，或者添加新的功能，他们就需要知道如何去做。是在GitHub上发起一个pull request？或者是在Bitbucket上？还是在邮件里发出一个补丁？所有的还是前面任何一个都可以？添加一个“如何贡献”的章节来说明一下你更偏向于如何接受代码。

一个“更新日志”章节可以帮助用户很快地去了解当他从版本X升级到版本Y的时候放生了哪些变动。并且，我强烈推荐你使用一个类似Semantic Versioning版本表格来描述你的插件，并且坚持这样做。你的用户会感谢你的。

最后，我喜欢添加一个“贡献列表”章节，这里会提到我的名字，并且列出其他所有对这个插件有启发意义的插件，需要感谢的贡献者，等等。

这是一个很好的出发点。对于一个独一无二的插件，你会感觉到需要从这个列表出发，并且这样做非常好。写文档没有什么比较难或者比较快的法则，除了以下几条：

- 要全面。
- 不要太啰嗦。
- 给用户一个从完全不会到成为插件专家的一个旅途。

目录

现在我们已经了解了我们会添加哪些章节，添加下面的内容到 `potion.txt` 文件里：

```
```

```
CONTENTS PotionContents
```

```
  1. Usage ................ |PotionUsage|
  2. Mappings ............. |PotionMappings|
  3. License .............. |PotionLicense|
  4. Bugs ................. |PotionBugs|
  5. Contributing ......... |PotionContributing|
  6. Changelog ............ |PotionChangelog|
  7. Credits .............. |PotionCredits|
```

```
```

这里有一些东西需要注意。首先，由 `=` 组成的行会被语法高亮。你可以用这些行来分割你文档的不同章节。你也可以使用 `-` 组成的行来分割子章节。

这里的 `*PotionContents*` 会创建另外一个标签，所以你可以用 `:help PotionContents` 直接定位到帮助文档的目录。

每个被 `|` 所围住的字符创建了一个到标签的链接。用户可以把光标定位到帮助文档的目录了。

Vim会隐藏这些 `*` 和 `|` 字符，然后对它们进行语法高亮，所以最后显示出来的是一个其他人可以用来进行查找的很好看的一个目录。

区块

你可以通过下面的方式来创建一个区块头：

```
```

```
Section 1: Usage PotionUsage

This plugin with automatically provide syntax highlighting for Potion files (files ending in .pn).

It also...
```

请确保你通过 `*` 字符创建了正确地标签，这样你的目录里的所有链接都能正常工作。

继续给你的目录你的每个章节创建一个区块头。

示例

我很想讲解一遍帮助文档的所有语法并且告诉你如何去使用它们，但是如何写帮助文档完全取决于个人的爱好。所以，我会给你一些有着很好文档的Vim插件作为示例。

对于每个产假，你可以把文档的源文件直接拷贝到你的Vim缓冲区，然后把它的文件类型设置为 `help` 来看看它会渲染成什么样的结果。当你想看具体是如何实现的时候，你可以通过切换回 `text` 的方式来查看。

你可以创建一个按键映射来创建一个可以把当前缓冲区的文件类型在在 `help` 和 `text` 之间进行切换的开关。

- Calm，我个人的一个配合Shell命令工作的插件。这是一个简单的用来展示我之前所提到的所有章节的示例。
- NERD tree，一个由Scrooloose创建的用来浏览文件的插件。注意它的文档结构，同时，注意他在具体介绍每个映射之前是如何先总结出一个更加易读的列表的。

- Surround，一个由Tim Pope创建的用来处理“包围”字符串的插件。注意它没有目录，，同时注意它的不同样式的区块头，同时注意他的表格列的头部。了解一下这些效果是如何实现的，然后觉得你是否喜欢他们。这个看个人的喜好。
- Splice，我个人用来解决在版本冲突中出现的合并冲突的三个文件的插件。注意这里的，映射列表的格式，同时留心我是怎么使用ASCII艺术字做的图表来解释布局的。有些时候真的是一图胜千言。

记住所有的Vim自身的文档都可以作为参考示例。你可以从中学习到很多。

开写！

现在你已经看过其他的插件是如何划分文档的结果，以及如何去写文档了。你可以开始给你的Potion插件写文档了。

如果你还不习惯写技术文档的花，那么这个对你而言可能就是一个挑战了。学习如何去写当然不是很简单，但是和其他许多技能一样，它只需要你去练习就可以了，所以直接开始写吧！你没必要写的非常完美，你以后可以一直来改善它。

不要害怕写一些你还不完全确定的东西，然后丢弃它们，最后再进行重写。经常在缓冲区放一些东西能够让你很快进入写的状态。如果你想回头再看看，它们还会留在版本控制里。

一个很好地开始的方式就是假设你有一个坐在你旁边的朋友，他也是在用Vim。他们没有哟过过你的插件，但是很感兴趣，你的目标是让他们变成使用你插件的专家。假设你是在给一个普通的人讲解，是一个很好地方式，它能够限制你在讲清楚整件事大体上是如何运作之前，不要变得太技术化。

如果你还是被卡住了，并且没有准备好为整个插件来编写文档，你可以尝试一些更小的东西。选择你的 `~/.vimrc` 里的一个映射，然后通过注释的方式提供完整的文档。解释一下它是用来干什么的，怎么使用，如何工作的。例如，以下是我自己的 `~/.vimrc` 文件：

```
" "Uppercase word" mapping.
"
" This mapping allows you to press  in insert mode to convert the
" current word to uppercase.  It's handy when you're writing names of
" constants and don't want to use Capslock.
"
" To use it you type the name of the constant in lowercase.  While
" your cursor is at the end of the word, press  to uppercase it,
" and then continue happily on your way:
"
"                            cursor
"                            v
"     max_connections_allowed|
"
"     MAX_CONNECTIONS_ALLOWED|
"                            ^
"                            cursor
"
" It works by exiting out of insert mode, recording the current cursor
" location in the z mark, using gUiw to uppercase inside the current
" word, moving back to the z mark, and entering insert mode again.
"
" Note that this will overwrite the contents of the z mark.  I never
" use it, but if you do you'll probably want to use another mark.
inoremap  mzgUiw`za
```

它比一个完整插件的文档要短的多，但是它是一个帮助你进行练习的一个很好的例子。如果你把 `~/.vimrc` 文件放到GitHub或者Bitbucket上的话，它也能够帮助别人去理解你的这个文件。

发布

现在你已经掌握了能够编写对很多人都有用的Vim插件了。这个章节会讲解如何如何把你的插件上线并且让他们能够很容易获得，就像推销一样。

放在主机上

你首先需要做得是，把你的插件放到网络上，这样其他的人就可以下载了。官方的放Vim插件的地址是Vim网站的脚本区块。

目前的一个趋势是，在Bitbucket或者GitHub这样的公告仓库上。这个方法变得流行可能主要因为两个因素。第一，Pathogen让插件能够很容易的安装在不同的目录下。类似Mercurial和Git的分布式版本控制系统，已经类似Bitbuck和GibHut的公共主机站点的兴起也对这有着影响。

对于很多把代码放在自己的版本控制库的人而言，提供代码库是一件很简单的事情。Mercurial用户可以使用Mercurial的“子仓库”来跟踪插件的版本，Git用户可以使用子模块（虽然是使用其他的仓库，而不是类似Mercurial的子仓库。

对于每个插件有个单独的代码库，当它们出现问题的时候，调试起来是很方便的。你可以使用blame，bisection等其他VSC提供的工具来排查究竟出了什么问题。而且，如果你本地有了代码，就很容易提供修复的代码了。

希望我说服了你，你需要让你的仓库能够被其他用户取到。你用什么服务并不重要，只要你的仓库对其他用户是可见的就可以了。

文档

你已经使用Vim内部文档的格式来给你的插件编写文档了，但是你的工作并没有借宿。你还需要写一个简要的说明来总结一些东西：

- 1. 你的插件是做什么的？
- 1. 这些用户为什么需要使用它？
- 1. 为什么你的插件比其他的更好用（如果有的话）？
- 1. 插件的许可证是什么？
- 1. 一个到完整文档的链接，这个文档是通过vim-doc渲染的。

这些内容应该放在你的README文档（这个会在你的Bitbucket和GitHub的仓库的首页展示），而且你可以把它用作你的插件在Vim.org上的入口描述上。

放一些截图也是一个很好地注意。虽然Vim是一个文本编辑器，但这并不代表它没有用户界面。

公开宣传

一旦你把你的插件放在网络上的这些位置上：现在可以告诉整个世界了！你可以在Twitter上告诉你的粉丝，发布到Reddit的/r/vim模块，在你的个人网站上挂上入口的链接，也可以在Vim初学者的邮件列表上发布它。

当你把你个人创造的作品发布到外界的时候，你会得到赞赏也会得到批评。不要让消极的文字影响你太多。仔细聆听他们，但是不要因为别人指出一些瑕疵（有效的或者其他的）而影响你的情绪。没人是完美的，这才是真实的互联网，所以如果你需要保持快乐和充满动力的话，那你就不要在意这些。

练习

在Vim.org上创建一个账号，如果你没有的话。

阅读一些你比较喜欢的插件的README文件，看看它们是如何组织的以及它们都包含了哪些信息。

文档

接下来干什么？

如果你阅读到这里了，并且完成了所有的示例和练习，你现在应该对Vimscript的基础知识有了很好的了解。别担心，还有很多东西需要学习！

颜色主题

在这本书里，我们给Potion文件加上了语法高亮。另外一个方面是定义自己的颜色主题，对于不同的元素定义不同的颜色。

颜色主题在Vim里如果要实现的话，还是是比较简单的，有点重复的。阅读 `:help highlight` 来学习一些基础内容。你可以看看一些内置的颜色主题，看看它们的文件结构是如何组织的。

如果你想挑战的话，你可以看看我自己的Bad Wolf的颜色主题，看看我是怎么使用Vimscript来让定义更加易于我自己进行维护的。注意里面的 `palette` 词典以及里面动态生成 `highlight` 命令的的 `HL` 函数。

Command命令

很多插件运行用户通过按键映射以及函数调用来进行交换，但是也有些喜欢创建Ex命令。例如，Fugitive插件创建类似 `:Gbrowse` 以及 `:Gdiff` 并且让用户来决定调用哪个。

类似这样的命令都是通过 `:comman` 命令创建的。阅读 `:help user-commands` 来学习如何创建你自己的命令。你现在已经了解了足够多的关于Vimscript的只是，阅读Vim的文档已经可以让你自己学习新的命令了。

运行时路径

在这本书里面，我对于Vim是如何决定加载那个文件只是简单地说了“使用Pathogen来管理”。现在你已经对Vimscript比较了解了，你可以阅读 `:help runtimepath`，同时阅读Pathogen的源码来了解究竟是如何实现的。

全方位补全

Vim提供了多种不同的补全文本的方式（阅读 `:help ins-completion` 来进行了解）。大部分都是很简单地，其中最强大的是 `全方位补全`，它可以让你调用一个自定义的Vimscript函数，来实现所有你想要进行补全的方法来进行补全。

如果你准备好了，要深入了解全方位补全这个兔子洞的内容的花，你可以开始阅读 `:help ominifunc` 以及 `:help coml-omni` 的内容，然后沿着里面的小路跟进去。

编译器支持

在我们的Potion插件里，我们创建了一些映射来编译和运行我们的Potion代码。Vim对于和编译器交互提供了更加深入的支持，包含解析编译错误，还提供了一个很好的能让你跳转到具体错误行的列表。

如果你对这个很感兴趣，你可以通过阅读 `:help quickfix.txt` 来深入了解。不过，我会警告你，你脆弱的心脏可能受不了 `errorformat` 的折磨！

其他的语言

这本书主要是关于Vimscript的，但是Vim也提供了一些其他语言的借款，例如Python，Ruby，以及Lua。也就是说，如果你不喜欢Vimscript的话，你也可以是用其他的语言。

但是了解Vimscript对于编辑你的 `~/.vimrc` 文件是有帮助的，同时它也能帮助你理解Vim提供给其他语言的API。但是使用其他的语言，可以帮你摆脱一些Vimscript中让人讨厌的东西，特别是对于大型插件而言。

如果你想了解使用某个语言来操作Vim，你可以阅读它们的文档：

- `:help Python`
- `:help Ruby`

- `:help Lua`
- `:help perl-using`
- `:help MzScheme`

Vim的文档

作为最后一部分内容，下面有一个Vim帮助的列表，这里有一些非常有帮助的，有很多有用信息的，有趣的或者仅供娱乐的（没有按照任何顺序排列）：

- `:help various-motions`
- `:help sign-support`
- `:help virtualedit`
- `:help map-alt-keys`
- `:help error-messages`
- `:help development`
- `:help tips`
- `:help 24.8`
- `:help 24.9`
- `:help usr_12.txt`
- `:help usr_26.txt`
- `:help usr_32.txt`
- `:help usr_42.txt`

练习

写一个你一直想做的插件，然后分享给整个世界！