

The
Pragmatic
Programmers

异步图书
www.epubit.com.cn

以思维的速度来编辑文本 畅销图书新版，根据Vim 7.4更新

Vim实用技巧

(第2版)

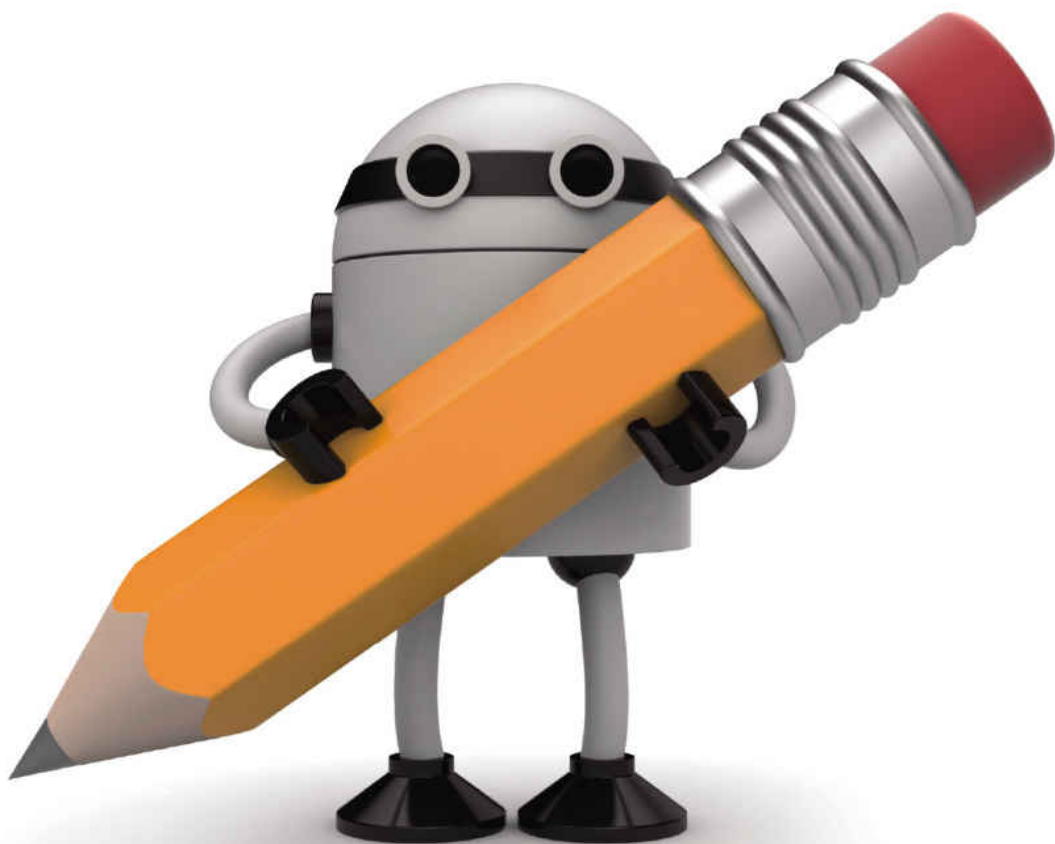
使用模式入门篇

Practical Vim

Edit Text at the Speed of Thought

[英] Drew Neil 著

杨源 车文隆 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

版权信息

书名：Vim实用技巧（第2版）

ISBN：978-7-115-42786-1

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

• 著 [英] Drew Neil

译 杨 源 车文隆

责任编辑 陈冀康

• 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

• 读者服务热线：(010)81055410

反盗版热线：(010)81055315

版权声明

Simplified Chinese-language edition Copyright © 2016 by Posts & Telecom Press. All rights reserved.

Copyright © 2015 The Pragmatic Programmers, LLC. Original English language edition, entitled Practical Vim, second Edition.

本书中文简体字版由The Pragmatic Programmers, LLC授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

目录

[版权信息](#)

[版权声明](#)

[内容提要](#)

[读者对本书的评论](#)

[第一版序](#)

[前言](#)

[致谢](#)

[写作体例说明](#)

[第1章 Vim解决问题的方式](#)

[技巧1 认识 . 命令](#)

[. 命令是一个微型的宏](#)

[技巧2 不要自我重复](#)

[减少无关的移动](#)

[技巧3 以退为进](#)

[使修改可重复](#)

[使移动可重复](#)

[合而为一](#)

[技巧4 执行、重复、回退](#)

[技巧5 查找并手动替换](#)

[偷懒的办法：无需输入就可以进行查找](#)

[使修改可重复](#)

[合而为一](#)

[技巧6 认识 . 范式](#)

[回顾前面3个 . 命令编辑任务](#)

[理想模式：用一键移动，另一键执行](#)

[第一部分 模式](#)

[第2章 普通模式](#)

[技巧7 停顿时请移开画笔](#)

[技巧8 把撤销单元切成块](#)

[技巧9 构造可重复的修改](#)

[反向删除](#)

[正向删除](#)

删除整个单词

决胜局：哪种方式最具重复性？

结论

技巧10 用次数做简单的算术运算

技巧11 能够重复，就别用次数

只在必要时使用次数

技巧12 双剑合璧，天下无敌

操作符 + 动作命令 = 操作

扩展命令组合的威力

自定义操作符与已有动作命令协同工作

自定义动作命令与已有操作符协同工作

第3章 插入模式

技巧13 在插入模式中可即时更正错误

技巧14 返回普通模式

结识插入-普通模式

技巧15 不离开插入模式，粘贴寄存器中的文本

对面向字符的寄存器使用 <C-r>{register} 命令

技巧16 随时随地做运算

技巧17 用字符编码插入非常用字符

技巧18 用二合字母插入非常用字符

技巧19 用替换模式替换已有文本

用虚拟替换模式替换制表符

第4章 可视模式

技巧20 深入理解可视模式

技巧21 选择高亮选区

激活可视模式

在可视模式间切换

切换选区的活动端

技巧22 重复执行面向行的可视命令

准备工作

先缩进一次，然后重复

技巧23 只要可能，最好用操作符命令，而不是可视命令

使用可视模式下的命令

使用普通模式下的操作符命令

结论

技巧24 用面向列块的可视模式编辑表格数据

[技巧25 修改列文本](#)

[技巧26 在长短不一的高亮块后添加文本](#)

[第5章 命令行模式](#)

[技巧27 认识Vim的命令行模式](#)

[Vim命令行模式中的特殊按键](#)

[Ex 命令影响范围广且距离远](#)

[技巧28 在一行或多个连续行上执行命令](#)

[用行号作为地址](#)

[用地址指定一个范围](#)

[用高亮选区指定范围](#)

[用模式指定范围](#)

[用偏移对地址进行修正](#)

[结论](#)

[技巧29 使用‘:t’和‘:m’命令复制和移动行](#)

[用‘:t’命令复制行](#)

[用‘:m’命令移动行](#)

[技巧30 在指定范围上执行普通模式命令](#)

[技巧31 重复上次的 Ex 命令](#)

[技巧32 自动补全 Ex 命令](#)

[在多个补全项间选择](#)

[技巧33 把当前单词插入命令行](#)

[技巧34 回溯历史命令](#)

[结识命令行窗口](#)

[技巧35 运行Shell命令](#)

[执行 Shell 中的程序](#)

[把缓冲区内容作为标准输入或输出](#)

[使用外部命令过滤缓冲区内容](#)

[结论](#)

[技巧36 批处理运行Ex命令](#)

[逐条执行Ex命令](#)

[把Ex命令存成脚本并加载](#)

[用此脚本修改多个文件](#)

[欢迎来到异步社区！](#)

内容提要

Vim是一款功能丰富而强大的文本编辑器，其代码补全、编译及错误跳转等方便编程的功能特别丰富，在程序员中得到非常广泛的使用。Vim能够大大提高程序员的工作效率。对于Vim高手来说，Vim可以与思考同步的速度编辑文本。同时，学习和熟练使用Vim又有一定的难度。

本书为那些想要提升自己的程序员编写，阅读本书是熟练掌握高超的Vim技巧的必由之路。全书共21章，包括123个技巧。每一章都是关于某一相关主题的技巧集合。每一个技巧都有针对性地解决一个或一类问题，帮助读者提升Vim的使用技能。本书示例丰富，讲解清晰，采用一种简单的标记方法，表示交互式的编辑效果，可以帮助读者快速掌握和精通Vim。

本书适合想要学习和掌握Vim工具的读者阅读，有一定Vim使用经验的程序员，也可以参考查阅以解决特定的问题。

读者对本书的评论

我通过本书学到的Vim知识比从其他渠道获得的多得多。

► Robert Evans 软件工程师，编码狂人

读完本书的几章后，我意识到自己有多么孤陋寡闻，在30分钟时间里一下子被从中级用户打回到初学者。

► Henrik Nyh 软件工程师

本书不断地改变我对一个编辑器能做什么的信仰。

► John P. Daigle 开发人员，ThoughtWorks 公司

Drew 在本书中继续了他在Vimcasts网站上的杰出工作。对任何关注 Vim 的人来说，这都是一本不可错过的书。

► Anders Janmyr 开发人员，Jayway

本书在官方文档和如何真正使用 Vim 之间架设了一座跨越鸿沟的桥梁。读完几章以后，我就把默认编辑器换成了 Vim，从此再未换过。

► Javier Collado 自动化QA工程师，Canonical公司

Drew Neil 远不止为我们展示了工作所用的正确工具。他于穿插叙述之中，揭示了每个决定背后的哲学。本书教你让Vim用手指思考，而不是期待你记住所有东西。

► Mislav Marohnic 顾问

我将 Vim 用于做服务器维护已经超过15年了，但只在最近才把它用于软件开发。我以为我了解 Vim，但本书极大地提升了我的编码效

率。

► Graeme Mathieson 软件工程师，Rubaidh公司

本书让我意识到对于 Vim 我还有多少东西要学。每个技巧都可以很容易地马上应用到工作过程当中，从多方面提升你的工作效率。

► Mathias Meyer 《Riak 手册》的作者

在Vim知识方面，本书是一个无尽的宝藏。现在我用 Vim 处理日常工作已经超过两年了，这本书给了我无穷的启发。

► Felix Geisendörfer 联合创始人，Transloadit

第一版序

传统观点认为，Vim的学习曲线很陡，但我相信绝大多数Vim用户对此不以为然。在学习Vim的初期，人们的确需要经历一段驼峰似的阻力，然而一旦完成了vimtutor的训练，并了解如何为vimrc配置一些基本选项后，就会达到一个新的高度，能用Vim完成实际工作了——尽管步履蹒跚，但终有回报。

接下来该做什么呢？来自互联网的答案是所谓的“技巧”——一种解决特定问题的灵丹妙药。当你觉得解决某个问题的方法不是最佳时，没准儿就要去搜索专门解决它的技巧了，或者你可能会主动看一些更受追捧的技巧。根据我的学习经验，这种策略的确奏效，不过这样学得很慢。“用*查找光标下的单词”这一招固然会让你受益匪浅，但却难以帮助你像Vim高手一样思考问题。

当我发现本书正是以这种“技巧”的方式组织章节时，你一定能理解我所持的怀疑态度。这区区上百条技巧怎么能让我举一反三呢？但当我翻了几页本书之后，我才意识到自己对“技巧”的理解太片面了。本书介绍的技巧与我认为是“问题 / 解决方法”方式有所不同，它旨在向人们传授如何像Vim高手一样思考问题。从某种意义上讲，这些技巧更像是寓言故事而非医师处方。书中的前几条技巧向人们介绍了应用范围很广的.命令，这是Vim高手们最重要的看家法宝，因为当时没人指点，我自己过了多年才意识到这一点。

正是由于这个原因，我才对本书的出版感到如此兴奋。如果现在再有Vim新手问我“下一步该学什么”，我知道该告诉他们什么了。不管怎么说，本书甚至还教会了我不少东西呢。

Tim Pope

Vim 核心贡献者

2012 年 4 月

前言

本书是为那些想提升自己的程序员写的。你一定听说过，对于Vim高手来说，Vim能以思考的速度编辑文本。阅读本书则是通往此途的必经之路。

本书是精通Vim的捷径。尽管它不会手把手教你，不过初学者可以先运行随Vim发布的交互式课程——Vim向导 [\[1\]](#) 来了解必备的知识。本书则在这一基础之上着重介绍核心概念，并为你讲解地道的用法。

Vim是高度可配置的，然而定制是一件很个性化的事情，因此我试图避免建议什么应该放进你的vimrc里，什么不应该。相反，本书关注的是Vim编辑器的核心功能。不管你是通过SSH登录远端服务器工作，还是在用本地安装了插件而增添了额外功能的GVim，这些功能都永远在那儿。精通了Vim的核心功能，你就获得了一个可移植的、强大的文本编辑工具。

本书如何组织

这是一本按技巧组织的书，它被设计成不必从头读到尾（没错！在下一章开头，我会建议你直接跳到正文）。每一章都是关于某一相关主题的技巧集合，而每个技巧都讲解一个特定的实用功能。有些技巧自成一体，而有些技巧则依赖本书中其他地方的内容，这些有依赖关系的技巧会以交叉引用的形式呈现给大家，因此你可以轻松找到所有内容。

虽然整本书的进度安排不是先从入门开始，然后再到高级，但是每个独立章节中的内容都是按循序渐进的方式来组织的。缺乏经验的Vim用户可能更愿意先浏览全书，只阅读每章的前几个技巧；而高级用户可能会重点看每章中比较靠后的技巧，或是根据需要查阅本书。

关于示例的说明

在Vim中，对一件给定的任务，总能找到不止一种解决办法。例如，第1章里的所有问题都围绕`.`命令进行设计，以便讲解`.`命令的应用，不过这些问题也都可以用`:substitute`命令解决。

在阅读我的解决方法时，你自己也许会想：“难道用这种方法做不是更快吗？”可能你是对的！我的解决方法只是在讲解一种特定的技术，试图透过它们简单的外表，找出它与你日常所面临问题的相似之处，而这一点才是这些技巧可以帮你节省时间的地方。

先学会盲打，然后再学习Vim

如果你要低头看着键盘打字，那学习Vim的好处不会立竿见影地显现出来。要高效地使用Vim，必须学会盲打。

Vim的祖先要追溯到经典的Unix编辑器vi和ed，参见技巧27中的“Vim（及其家族）的词源”部分，它们比鼠标及所有点击界面出现得都早，因此根本没有这类接口，所有操作都通过键盘完成。Vim也是一样，Vim中的所有操作也都可以通过键盘完成。对盲打人员来说，这意味着用Vim做任何事都能更快些。

[1] http://vimhelp.appspot.com/usr_01.txt.html#vimtutor

致谢

感谢Bram Moolenaar 创造了 Vim，也感谢所有对 Vim 的开发做出贡献的人。这是一个永恒的软件，我期盼自己能与它一同成长。

感谢Pragmatic Bookshelf 公司的每个人，正是你们的齐心协力才使这本书变得更好。特别感谢本书的项目编辑Kay Keppler，感谢他教导我成为一名作者，并促使本书成形，而不在乎经历了多少的成长之痛以及我偶尔使性子。同样要感谢本修订版的项目编辑Katharine Dvorak。我也想感谢David Kelly，感谢他对我与众不同的排版要求所做出的巧妙设计。

本书刚开始并没有打算按照技巧的方式组织章节，但Susannah Pfalzer 发现采用这种格式会更好。重写这么多内容的确很痛苦，但做完之后，我头一次写出了让自己满意的初稿。Susannah 知道什么是最好的，感谢她分享了这一见解。

感谢Dave Thomas 和 Andy Hunt 创办了Pragmatic Bookshelf 公司。我没想过让其他出版商出版本书，并且我很荣幸本书能和其他书一起列在他们的书目中。

如果没有技术审阅人员，本书不可能出版。每一位技术审阅人都为之贡献了力量，并帮助本书成形。在此我想感谢Adam McCrea、Alan Gardner、Alex Kahn、Ali Alwasity、Anders Janmyr、Andrew Donaldson、Angus Neil、Charlie Tanksley、Ches Martin、Daniel Bretoi、David Morris、Denis Gorin、Elyézer Mendes Rezende、Erik St. Martin、Federico Galassi、Felix Geisendörfer、Florian Vallen、Graeme Mathieson、Hans Hassel- berg、Henrik Nyh、Javier Collado、Jeff Holland、Josh Sullivan、Joshua Flanagan、Kana Natsuno、Kent Frazier、Luis Merino、Mathias Meyer、Matt Southerden、Mislav Marohnic、Mitch Guthrie、Morgan Prior、Paul Barry、Peter Aronoff、Peter Rihn、Philip Roberts、Robert Evans、Ryan Stenhouse、Steven、Ragnarök、Tibor Simic、Tim Chase、Tim Pope、Tim Tyrrell以及Tobias Sailer。

即便是有了技术审阅人员的审查，一些错误依然隐藏在书中。在此我想感谢为本书提交错误报告的所有人，是你们帮助我找到错误并修正它们。

Vim的内置文档是非常了不起的资源，本书中到处都在引用它。我想感谢Carlo Teubner把Vim文档在线发布到vimhelp.appspot.com，并持续更新。

本书第一版中有些技巧很糟糕，但我依然把它们加入书中，因为我觉得它们很重要。在本修订版中，我很高兴能够重写这几个糟糕的技巧。感谢Christian Brabandt实现了“改变玩法”的gn命令，让我能够重写技巧84。感谢Yeggapan Lakshmanan实现了cfdo命令（及其相关命令），让我能够重写技巧97。我也想感谢David Bürgin所提交的Vim补丁7.3.850，这一补丁终结了vimgrep命令带给我的小头疼。

另外，我想感谢整个 Vim 社区，感谢他们通过互联网分享见解。通过阅读StackOverflow 上 Vim 标签中的内容和订阅 vim_use 邮件列表，我学到了本书中的很多技巧。

Tim Pope的rails.vim插件对于说服我皈依 Vim 起了很大的作用，并且他开发的许多其他插件也都成为我的Vim设置中必不可少的组成部分。我也从Kana Natsuno的插件中领悟到很多东西，在我印象里，他的自定义文本对象是对Vim核心功能最好的扩展。感谢你们两位把Vim这把“锯子”磨得更加锋利，使我们大家从中获益。

感谢Joe Rozner 提供的 wakeup 源码，我使用它来介绍 :make 命令；感谢Oleg Efimov对nodeline缺陷的快速反馈，也感谢Ben Cormack对robots以及ninjas的解释。

2012 年 1 月，我们搬到了柏林，在这里的技术社区的启发下，我完成了本书。我想感谢Gregor Schmidt 成立了 Vim 柏林用户组，也感谢Schulz-Hofen 举办的招待我们的聚会。与 Vim 用户交谈的机会，真正帮我理清了思路，因此我感激每个参加 Vim 柏林会议的人。也感谢Daniel 及 Nina Holle 把自己的房子转租给我们，它既是居住的绝佳场所，也是工作的高效环境。

2011 年 3 月我住在埃及时，需要动手术清除肠道粘连。不幸的是，我离家很远；幸运的是，我妻子陪伴在身边。Hannah把我送到半岛南部医院，在那里我受到了很好的照顾。我想对那里的所有医务人员表示感谢，感谢他们对我的悉心照料，也感谢 Shawket Gerges 医生为我成功地进行了手术。


当我母亲知道我要动手术时，她抛下一切乘坐最早航班飞到埃及。要知道当时这个国家正在发生革命，她老人家需要多大的勇气才能做到这一切啊。我无法想象没有我母亲的支持与经验，我和Hannah该如何度过这段困难时期。我为一生中拥有两个如此伟大的女人而感到幸福。

写作体例说明

在本书中，我会通过图例进行演示，而不是描述它们，因为用书面语不太容易做到这一点。为了显示在交互式编辑会话中所采取的步骤，我采用了一种简单的标记方法，把按键操作及 Vim 缓冲区的内容排在一起进行说明。

如果你急于动手操作的话，现在可以直接跳过这一部分。这一部分主要描述本书沿用的体例，你会发现其中很多都不需要解释。或许在某个地方你会偶然发现一个符号，想搞清楚它究竟代表什么意思。当这种情况发生时，你可以回到这一部分来寻求答案。

了解Vim的内置文档

了解Vim文档的最好方式是花点时间阅读。我在书中给出了Vim文档入口的“超链接”，以方便读者找到相关文档。例如，这里有一个通往Vim向导的“超链接”：`:h vimtutor` 。

这个图标具有两个作用。第一，它起到指示牌的作用，把目光吸引到这些有用的参考信息上；第二，如果你在联网的电子设备上阅读本书的话，那么你可以单击这些图标，它会把你带到 Vim 在线文档的相应入口。从这个意义上讲，它的确是超链接。

但是，如果你正在阅读纸版书，那该怎么做？别担心，如果在你的手边有可访问的 Vim 程序，简单地输入图标前的命令即可。

例如，你可以输入 `:h vimtutor`（`:h` 是 `:help` 命令的简写）。你可以把它想成 `vimtutor` 文档的唯一地址，即某种形式的 URL。从这个意义上讲，此 `help` 引用也是一种指向 Vim 内置文档的超链接。

在书页中模拟Vim的标记

Vim 区分模式的界面把它同其他文本编辑器区别开来。以音乐作个比喻，让我们拿 Qwerty 键盘与钢琴键盘进行比较。一个钢琴家可以

每次只弹一个琴键来演奏主旋律，他也可以一次弹多个键来演奏和弦。对于多数文本编辑器，要触发一个键盘快捷键，需要先按住一个或多个修饰键，如控制键或命令键，然后再按另外一个键，在Qwerty键盘上的这种操作方式，等同于在钢琴键盘上演奏和弦。

某些Vim命令也由演奏和弦的方式触发。不过普通模式命令则被设计成输入一串按键。在Qwerty键盘上的这种操作方式，则等同于在钢琴键盘上演奏主旋律。

Ctrl-s 是用来表示组合键命令的惯用约定，意为“同时按控制键及 s 键”，但这种约定方式并不适合用来描述Vim区分模式的命令集。在本节，我们将结识贯穿于全书的标记，在讲解Vim的用法时会用到它们。

演奏主旋律

在普通模式中，我们按次序输入一个或多个键组成一条命令。这些命令看起来像下面这样：

| 标记 | 含义 |
|------------|------------------------------------|
| x | 按一次 x |
| dw | 依次按 d 、 w |
| dap | 依次按 d 、 a 、 p |

这些序列大多数包含两个或3个按键，但有的命令会更长。解读Vim 普通模式命令序列的含义可能颇具挑战性，不过经过练习后你会做得更好。

演奏和弦

当你看到诸如 `<C-p>` 这样的键时，它的意思不是“先按 `<`，然后按 `C`，再按 `-`，等等”。`<C-p>` 标记等同于 `Ctrl-p`，意为“同时按 `<Ctrl>` 及 `p`”。

我不会无缘无故地选择这种标记方式的。首先，在 Vim 的文档中使用了这种标记（`:h key-notation` ⓘ），我们也用它定义自定义按键映射项。另外，某些 Vim 命令由组合键及其他键以一定的次序组合在一起，这种标记也可以很好地表达这些命令。请看下面这些例子。

| 标记 | 含义 |
|-------------------------------------|--|
| <code><C-n></code> | 同时按 <code><Ctrl></code> 和 <code>n</code> |
| <code>g<C- ></code> | 按 <code>g</code> ，然后同时按 <code><Ctrl></code> 和 <code> </code> |
| <code><C-r>0</code> | 同时按 <code><Ctrl></code> 和 <code>r</code> ，然后按 <code>0</code> |
| <code><C-w><C-=></code> | 同时按 <code><Ctrl></code> 和 <code>w</code> ，然后同时按 <code><Ctrl></code> 和 <code>=</code> |

占位符

很多 Vim 命令需要以一定的次序按两个或多个按键。有些命令后面必须跟某种特定类型的按键，而其他命令后面则可以跟键盘上的任意键。我使用花括号表示一条命令后可以跟有效按键集合。下面是一些例子。

| 标记 | 含义 |
|----------------------|---------------------------|
| <code>f{char}</code> | 按 <code>f</code> ，后面跟任意字符 |

| 标记 | 含义 |
|------------------------------|--|
| `{a-z} | 按 ` ，后面跟任意小写字母 |
| m{a-zA-Z} | 按 m ，后面跟任意小写或大写字母 |
| d{motion} | 按 d ，后面跟任意动作命令 |
| <C-r>{register} | 同时按 <Ctrl> 和 r ，后面跟一个寄存器地址 |

显示特殊按键

有些特殊按键以其名字表示，下表节选了其中的一些。

| 标记 | 含义 |
|----------------------|---|
| <Esc> | 按退出键 |
| <CR> | 按回车键，也写作 <Enter> |
| <Ctrl> | 按控制键 |
| <Tab> | 按制表键 |
| <Shift> | 按切换键 |
| <S-Tab> | 同时按 <Shift> 和 <Tab> |

| 标记 | 含义 |
|---------------------|-------|
| <Up> | 按上光标键 |
| <Down> | 按下光标键 |
| _ | 按空格键 |

注意，空格由_表示。它和 `f{char}` 命令组合在一起时记为 `f_`。

区分不同模式下的输入

在操作Vim时，经常会从普通模式切换到插入模式，然后再切换回普通模式。Vim中的每个键都可能具有不同的含义，这取决于当前哪个模式生效。我用了另一种样式表示在插入模式中输入的键，这可以让人很容易地把它们与普通模式下的按键区分开来。

看看这个例子 `cwreplacement<Esc>`。普通模式命令 `cw` 会删除从光标位置到当前词结尾处的文本，并切换到插入模式。然后我们在插入模式中输入单词“**replacement**”，并按 `<Esc>` 键再切换回普通模式。

普通模式所用的样式也用于可视模式，而插入模式的样式也用来表示命令行模式及替换模式下输入的按键。你可以通过上下文清楚地知道当前处于哪个模式。

在命令行中操作

在有些技巧中，我们会在 `shell` 或 `Vim` 中执行一条命令行命令。下面是在 `shell` 中执行 `grep` 命令的格式。

```
⇒  
  
$ grep -n Waldo *
```

下面是执行 `Vim` 内置的 `:grep` 命令的格式。

```
⇒ :grep Waldo *
```

在全书中，`$` 符号表示在外部 `shell` 中执行一条命令行命令，`:` 提示符则表示这条命令在内部的命令行模式中执行。有时我们也会看到其他的提示符，包括：

| 提示符 | 含义 |
|-----|------------------------------------|
| \$ | 在外部 <code>shell</code> 中执行命令行命令 |
| : | 用命令行模式执行一条 <code>Ex</code> 命令 |
| / | 用命令行模式执行正向查找 |
| ? | 用命令行模式执行反向查找 |
| = | 用命令行模式对一个 <code>Vim</code> 脚本表达式求值 |

无论何时你在文中见到一条 `Ex` 命令，比如 `:write`，都可以假设我们按了 `<CR>` 键来执行该命令，否则该命令什么也不会做，因此可以认为 `<CR>` 在 `Ex` 命令中是隐含的。

与之相反，Vim 的查找命令允许在按 <CR> 前预览第一个匹配项（参见技巧82）。当你在文中见到一条查找命令时，比如 /pattern<CR>，你会看到 <CR> 键被显式地标出来了；如果 <CR> 被省略了，那是有意为之，也就是说你现在还不要按回车。

显示缓冲区内光标的位置

在显示缓冲区内容时，如果能指示当前光标位于何处，那会很有用。在下面的例子里，你可以看到光标位于单词“**One**”的第一个字母上。

```
O
ne two three
```

当我们执行一项包含若干步的修改时，缓冲区的内容会经历一些中间状态。为了讲解这一过程，我使用了一个表格，在其左栏中显示所执行的命令，在右栏中显示缓冲区的内容。下面是个简单的例子。

| 按键操作 | 缓冲区内容 |
|-----------|-----------------------|
| {start} | O ne two three |
| dw | t wo three |

在第2行，我们运行dw 命令删除了光标下的单词。通过查看位于同一行的缓冲区内容，我们可以立刻看到这条命令执行完后缓冲区的状态。

高亮显示查找匹配项

在讲解 Vim 的查找命令时，如果能把缓冲区内出现的每个匹配项都高亮显示出来，那会很有帮助。在下例中，查找字符串“the”会让出

现该模式的4处地方被高亮显示出来：

| 按键操作 | 缓冲区内容 |
|----------|---|
| {start} | t he problem with these new recruits is that they don't keep their boots clean. |
| /the<CR> | the problem with the se new recruits is that the y don't keep the ir boots clean. |

你可以跳到技巧81，了解如何激活 Vim 的查找高亮功能。

在可视模式中选择文本

可视模式允许在缓冲区内选择文本，然后在其上操作。在下例中，用it 文本对象选中 <a> 标签内的文本。

| 按键操作 | 缓冲区内容 |
|---------|--|
| {start} | <a h ref="http://pragprog.com/dnvm/">Practical Vim |
| vit | Practical Vim |

注意，高亮显示可视选区的样式与高亮显示查找匹配项的样式相同。当你看到这种样式时，根据上下文就可以知道它究竟是代表一处查找匹配项，还是一个高亮选区。

下载本书中的示例

本书中的例子通常都先显示修改前的文件内容，并在示例文本中给出该文件所在的路径，如下所示：

macros/incremental.txt

```
partridge in a pear tree  
turtle doves  
French hens  
calling birds  
golden rings
```

每当你看到以这种方式列出的文件路径时，都表示该例可被下载。我建议你**Vim**中打开此文件，然后亲自试试这个例子。这是学习**Vim**的最好方式。

要照着书中的例子操作，你可以从**Pragmatic Bookshelf**的网站上下载本书所有的示例和源代码 [\[1\]](#)。如果你在联网电子设备上阅读本书，也可以单击文件名来逐一获取每个文件。你可以用上面的例子试验一下。

使用**Vim**的出厂配置

Vim是高度可配置的，如果你不喜欢其默认的行为，可以改变它们。这本是好事，但是，如果你用自定义的**Vim**跟着做本书中的例子，可能会感到迷惑，你也许会发现有些东西并不像书中描述的那样工作。如果你怀疑是自定义配置造成了干扰，那么你可以做一个快速的测试。试着先退出**Vim**，然后再用下列选项启动它。

```
⇒  
  
$ vim -u NONE -N
```

`-u NONE` 标志让**Vim**在启动时不加载你的**vimrc**，这样，你的定制项就不会生效，插件也会被禁用。当用不加载**vimrc**文件的方式启动时，**Vim**会切换到**vi**兼容模式，这将导致很多有用的功能被禁用，`-N`标志则会使能 `'nocompatible'` 选项，防止进入**vi**兼容模式。

对于本书中的大多数例子来说，用 `vim -u NONE -N` 启动 Vim 应该可以确保你获得与书中的描述相符的体验，不过也有几处例外。有些 Vim 的内置功能是由 Vim 脚本实现的，也就是说，只有在激活插件时，它们才会工作。下面的文件中包含了激活 Vim 内置插件的最小配置。

essential.vim

```
set
    nocompatible
filetype
    plugin on
```

在启动 Vim 时，可以执行如下命令，用该文件取代你的 `vimrc`。

```
⇒
$ vim -u code/essential.vim
```

在执行时，需要相应地调整 `code/essential.vim` 文件所在的路径。激活 Vim 内置的插件功能后，可以使用诸如 `netrw`（参见技巧 44）、`omni-completion`（参见技巧 119），以及很多其他的功能。我在本书中所说的 Vim 的出厂配置，指的就是激活了内置的插件功能，并且禁用了 `vi` 兼容模式时的配置。

需要留意技巧开头的名为“准备工作”的小节，要想跟着技巧中的步骤做，需要对 Vim 进行相应的配置。如果你由 Vim 的出厂配置开始，然后再动态应用这些定制项，就应该能重现技巧中的结果，不会遇到任何问题。

如果你仍遇到问题，请看后面的“关于 Vim 的版本”部分。

Vim 脚本所扮演的角色

Vim脚本让我们可以给Vim添加新的功能，或是改变其已有的功能。它是一种完整的脚本语言，并且这个主题本身就可以写一整本书。不过本书并不是这样一本书。

但我们不会完全避开此话题，Vim脚本一直隐身在幕后，时刻准备响应我们的召唤。在技巧16、技巧71、技巧95及技巧96中，我们将看到一些如何使用它们完成日常工作的例子。

本书展示了如何使用Vim的核心功能。换句话说，它假设我们不使用任何第三方插件。不过技巧87是个例外，`visual-star.vim` 插件添加的功能我认为是不可或缺的，并且它只需很少的代码——不超过10行。同时它也展示了扩充Vim的功能是多么容易。文中给出了`visual-star.vim`的实现，但没有讲解。这应该能给你一些印象，了解Vim脚本是什么样的，以及你能用它干什么。如果它激起了你的兴趣，那就更好了。

关于 Vim 的版本

本书中的所有例子都在最新的Vim版本中测试过，在写本书时是版本7.4。就是说，大多数例子在任意7.x版本中都能够很好地工作，并且所讨论的很多功能在6.x中也同样适用。

有些Vim功能可以在编译期间被禁用。例如，在配置编译选项时，可以传入`--with-features=tiny`参数，这会禁用除最基本的功能外的其他所有功能（Vim的功能集还包括`small`、`normal`、`big`和`huge`）。可以查阅`:h +feature-list` ❶，浏览完整的功能列表。

如果你发现自己的Vim缺少本书所讨论的某个功能，那么你也许正在使用一个最小功能集的Vim发行版。可以用`:version`命令检查此功能是否可用。

```
⇒ :version
```

```
《  VIM - Vi IMproved 7.4 (2013 Aug 10, compiled Oct 14 2015
```

```
18:41:08)
  Huge version without GUI. Features included (+) or not (-):
  +arabic +autocmd +balloon_eval +browse +builtin_terms
+byte_offset
+cindent +clientserver +clipboard +cmdline_compl +cmdline_hist
+cmdline_info +comments
  ...
```

在现代计算机上，没理由不用 Vim 的 huge 功能集！

用终端 Vim 还是图形化Vim？你自己定！

传统上，Vim在终端内运行，没有图形用户界面（GUI）。我们也可以说Vim具有TUI，即文本用户界面。如果你每天有大量时间花在命令行上，你会感觉这很自然。

如果你通常使用基于图形用户界面的文本编辑器，那么GVim（或OSX上的MacVim）可以给你提供一个通往Vim世界的有用桥梁（参见：[:h gui](#) ①）。GVim支持更多的字体以及更多的语法高亮颜色，也可以使用鼠标。它也遵从某些操作系统的约定，例如，在MacVim中，可以用Cmd-X 和Cmd-V 与系统剪切板交互，也可以用 Cmd-S 保存文件，用 Cmd-W 关闭一个窗口。如果你能接受的话，可以用这些命令，不过你应该已意识到，还有更好的方法完成这些。

对本书的目的而言，运行终端Vim还是GVim关系不大。我们将着重于介绍Vim的核心命令，这些功能在两者中都能很好地运行。我们要学习的重点是如何用Vim的方式来工作。

[1] http://pragprog.com/titles/dnvim/source_code

第1章 Vim解决问题的方式

从本质上讲，我们的工作具有重复性。不论是在几个不同的地方做相同的小改动，还是在文档的相似结构间移动，我们都会重复很多操作。凡是简化重复性操作的方式，都会成倍地节省我们的时间。

Vim对重复性操作进行了优化。它之所以能高效地重复，是因为它会记录我们最近的操作，让我们用一次按键就能重复上次的修改。这听起来很强大，但是除非我们能够学会规划按键动作，使得在重复时能完成一项有用的工作，否则这没什么用。掌握这一理念是高效使用Vim的关键。

我们将以`.`命令作为开始。这个看似简单的命令是Vim中的瑞士军刀，掌握它的用法是精通Vim的第一步。我们将运行一些可由`.`命令快速完成的简单编辑任务，虽然每个任务彼此之间截然不同，但解决的方法却大同小异。我们将找到一种理想的编辑模式，即用一次按键移动，用另一次按键执行。

技巧1 认识`.`命令

`.`命令可以让我们重复上次的修改，它是Vim中最为强大的多面手。

Vim文档只是简单地提到`.`命令会“重复上次修改”（参见`:h.`①），这听起来没什么特别，但在这个简单的说明里，我们会发现让Vim区分模式的编辑模型如此高效的核心原因。首先我们要问：“究竟什么是修改？”

要理解 . 命令的强大，我们需要意识到这一点：“上次修改”可以指很多东西，一次修改的单位可以是字符、整行，甚至是整个文件。

我们将使用下面这段文本进行说明。

the_vim_way/0_mechanics.txt

```
Line one
Line two
Line three
Line four
```

x 命令会删除光标下的字符，在这种情况下使用 . 命令“重复上次修改”时，就会让Vim删除光标下的字符。

| 按键操作 | 缓冲区内容 |
|---------|------------|
| {start} | L ine one |
| | Line two |
| | Line three |
| | Line four |
| x | ine one |
| | Line two |
| | Line three |
| | Line four |

| 按键操作 | 缓冲区内容 |
|------|------------|
| . | n e one |
| | Line two |
| | Line three |
| | Line four |
| .. | one |
| | Line two |
| | Line three |
| | Line four |

我们可以输入几次 u 撤销上述修改，使文档恢复到初始状态。

dd 命令也做删除操作，但它会把整行一起删掉。如果在 dd 后使用 . 命令，那么“重复上次修改”会让Vim删除当前行。

| 按键操作 | 缓冲区内容 |
|---------|-----------|
| {start} | L ine one |
| | Line two |

| 按键操作 | 缓冲区内容 |
|-----------|--------------------|
| | Line three |
| | Line four |
| dd | L ine two |
| | Line three |
| | Line four |
| . | L ine three |
| | Line four |

最后，>G 命令会增加从当前行到文档末尾处的缩进层级。如果在此命令后使用 . 命令，那么“重复上次修改”会让 Vim 增加从当前行到文档末尾的缩进层级。在下例中，让光标从第二行开始，以便一目了然地看出差别。

| 按键操作 | 缓冲区内容 |
|---------|------------------|
| {start} | Line one |
| | L ine two |
| | Line three |

| 按键操作 | 缓冲区内容 |
|------|------------|
| | Line four |
| >G | Line one |
| | Line two |
| | Line three |
| | Line four |
| j | Line one |
| | Line two |
| | Line three |
| | Line four |
| . | Line one |
| | Line two |
| | Line three |
| | Line four |
| j. | Line one |

| 按键操作 | 缓冲区内容 |
|------|------------|
| | Line two |
| | Line three |
| | Line four |

`x`、`dd` 以及 `>` 命令都是在普通模式中执行的命令，不过，每次进入插入模式时，也会形成一次修改。从进入插入模式的那一刻起（例如，输入 `i`），直到返回普通模式时为止（输入 `<Esc>`），Vim 会记录每一个按键操作。做出这样一个修改后再用 `.` 命令的话，它将会重新执行所有这些按键操作（参见技巧8中的 在插入模式中移动光标会重置修改状态 部分中的补充说明）。

. 命令是一个微型的宏

在第11章“宏”中，我们将看到Vim可以录制任意数目的按键操作，然后在以后重复执行它们。这让我们可以把最常重复的工作流程录制下来，并用一个按键重放它们。可以把 `.` 命令当成一个很小的宏（macro）。

我们将在本章看到一些关于 `.` 命令的应用，另外还将在技巧9及技巧23中学到 `.` 命令的一些最佳应用技巧。

技巧2 不要自我重复

对于在行尾添加内容这样的常见操作，如添加分号，Vim 提供了一个专门的命令，可以把两步操作合并为一步。

假设有如下的JavaScript程序片段。

the_vim_way/2_foo_bar.js

```
var
    foo = 1
var
    bar = 'a
'
var
    foobar = foo + bar
```

我们想在每行的结尾添加一个分号。要实现这一点，先得把光标移到行尾，然后切换到插入模式进行修改。`$` 命令可以完成移动动作，接着就可以执行 `a ;<Esc>` 完成修改了。

要完成全部修改，也可以对下面两行做完全相同的操作，不过那样做会错过这里将要提到的小窍门。由于 `.` 命令可以重复上次的修改，因此不必重复之前的操作，而是执行两次 `j$.`。一个键（`.`）顶3个（`a ;<Esc>`），虽然每次省得并不多，不过在重复操作时，累积效应可不小。

不过让我们再仔细审视一下这个操作模式：`j$.`。`j` 命令使光标下移一行，而 `$` 命令把光标移到行尾。我们用了两下按键，仅仅是为了把光标移到指定位置，以便可以用 `.` 命令。你觉得还有改进的余地吗？

减少无关的移动

`a` 命令在当前光标之后添加内容，`A` 命令则在当前行的结尾添加内容。不管光标当前处于什么位置，输入 `A` 都会进入插入模式，并把光标移到行尾。换句话说，它把 `$a` 封装成了一个按键操作。在本技巧后的一箭双雕部分中，我们将会看到Vim提供了不少这样的复合命令。

下面是对之前例子的改进。

| 按键操作 | 缓冲区内容 |
|------------|-------------------------------------|
| {start} | <code>v ar foo = 1</code> |
| | <code>var bar = 'a'</code> |
| | <code>var foobar = foo + bar</code> |
| A ; | <code>var foo = 1;</code> |
| | <code>var bar = 'a'</code> |
| | <code>var foobar = foo + bar</code> |
| j | <code>var foo = 1;</code> |
| | <code>var bar = 'a '</code> |
| | <code>var foobar = foo + bar</code> |
| . | <code>var foo = 1;</code> |
| | <code>var bar = 'a';</code> |
| | <code>var foobar = foo + bar</code> |
| j. | <code>var foo = 1;</code> |

| 按键操作 | 缓冲区内容 |
|------|--------------------------------------|
| | <code>var bar = 'a';</code> |
| | <code>var foobar = foo + bar;</code> |

用 A 来代替 \$a，大大提升了 . 命令的效率。不必再把光标移到行尾，只需保证它位于该行内就行了（可在任意位置）。现在可以重复执行足够多次的 j.，完成对后续行的修改。

一键移动，另一键操作，真是太完美了！请留意这种应用模式，因为我们即将在更多的例子中看到它的身影。

虽然这一模式对这个简短的例子来说很好用，但它不是万能的。试想一下，如果我们不得不给连续50行添加分号，即便每个修改输一次 j.，看起来也是一项很繁重的工作。跳到技巧30可以看到另外一种解决方法。

一箭双雕

我们可以这样说，A 命令把两个动作（\$a）合并成了一次按键。不过它不是唯一一个这样的命令，很多Vim的单键命令都可以被看成两个或多个其他命令的组合。下表列出了类似的一些例子，你能找出它们之间别的共同点吗？

| 复合命令 | 等效的长命令 |
|----------|------------|
| C | c\$ |
| s | cl |

| 复合命令 | 等效的长命令 |
|----------|------------|
| S | ^C |
| I | ^i |
| A | \$a |
| o | A |
| O | ko |

如果你发觉自己正在输入 **ko**（或更糟糕，在用 **k\$a**），马上打住！想想你在干什么，然后你就会意识到可以把它换成**O**命令。

你找出这些命令别的共同点了吗？它们全都会从普通模式切换到插入模式。仔细想想这一点，并想想这对 **.** 命令可能产生怎样的影响。

技巧3 以退为进

我们可以用一种常用的Vim操作习惯在一个字符前后各添加一个空格。乍一看，这种方法有点古怪，不过其好处是可重复，这将使我们可以事半功倍地完成工作。

假设有一行代码看起来是这样的：

the_vim_way/3_concat.js

```
var
    foo = "method
("+argument1+", "+argument2+")";
```

在JavaScript里把字符串连接到一起从来都不美观，但可以像下面这样在 + 号前后各添加一个空格，让肉眼更容易识别。

```
var
    foo = "method
(" + argument1 + ", " + argument2 + ")";
```

使修改可重复

下面的惯用方法可以解决这个问题。

| 按键操作 | 缓冲区内容 |
|---------|---|
| {start} | v ar foo = "method("+argument1+", "+argument2+")"; |
| f+ | var foo = "method("+ argument1+", "+argument2+")"; |
| s _+_ | var foo = "method(" + argument1+", "+argument2+")"; |
| ; | var foo = "method(" + argument1+ ", "+argument2+")"; |
| . | var foo = "method(" + argument1 + ", "+argument2+")"; |

| 按键操作 | 缓冲区内容 |
|------|---|
| ;. | var foo = "method(" + argument1 + "," + argument2+"); |
| ;. | var foo = "method(" + argument1 + "," + argument2 + "); |

s 命令把两个操作合并为一个：它先删除光标下的字符，然后进入插入模式。在删除 + 号后，先输入 `_+_`，然后退出插入模式。

先后退一步，然后前进三步，这是个奇怪的小花招，看起来可能不够直接。但这样做最大的好处是：我们可以用 `.` 命令重复这一修改。我们所要做的只是把光标移到下一个 + 号处，然后用 `.` 命令重复这一操作即可。

使移动可重复

本例中还有另外一个小窍门。`f{char}` 命令让 Vim 查找下一处指定字符出现的位置，如果找到了，就直接把光标移到那里（参见 `:h f` ❶）。因此，输入 `f+` 时，光标会直接移到下一个 + 号所在的位置。我们将会在技巧50里学到更多关于 `f{char}` 命令的知识。

完成第一处修改后，可以重复按 `f+` 命令跳到下一个 + 号所在的位置。不过，还有一种更好的方法可以用。`;` 命令会重复查找上次 `f` 命令所查找的字符，因此不用输入4次 `f+`，而是只输入一次，后面跟着再用3次 `;` 命令。

合而为一

`;` 命令带我们到下一个目标字符上，`.` 命令则重复上次的修改。因此，可以连续输入3次 `;.` 来完成全部修改。看起来是不是很熟悉？

与其和 Vim 区分模式的编辑模型做斗争，倒不如与它一起协同工作。然后，你就会发现它能把特定任务变得多么的容易。

技巧4 执行、重复、回退

在面对重复性工作时，我们需要让移动动作和修改都能够重复，这样就可以达到最佳编辑模式。Vim对此的支持是：它会记住我们的操作，并使最常用的操作触手可及，所以可以很方便地重复执行它们。本节将介绍 Vim 可以重复执行的每个操作，并学习如何回退这些命令。

我们已经看到 `.` 命令会重复上次修改。由于很多操作都被当成一次修改，因此 `.` 命令已经证明了它的神通广大。但有些命令能以其他的方式重复。例如，`@:` 可以用来重复任意 Ex 命令（在技巧31中讨论），或者也可以输入 `&`（参见技巧93）来重复上次的 `substitute` 命令（它本身也是一条 Ex 命令）。

如果我们知道如何重复之前的操作，而无需每次都输入整条命令，那么就会获得更高的效率。可以先执行一次，随后只需重复即可。

然而，这么少的按键就可以完成这么多的事情，这也可能会带来麻烦。我们需要很小心地操作才行，不然就很容易出错。当一遍又一遍地连续按 `j.j.j.` 时，那种感觉就像是在敲鼓。可是，如果不小心在一行上敲了两次 `j` 键，会发生什么？或是更糟，敲了两次 `.` 键？

当 Vim 让一个操作或移动可以很方便地重复时，它总是会提供某种方式，让我们在不小心做过头时能回退回来。对 `.` 命令而言，我们永远可以按 `u` 键撤销上次的修改。如果在使用 `f{char}` 命令后，不小心按了太多次 `;` 键，就会偏离我们的目标。不过可以再按 `,` 键跳回去，这个命令会反方向查找上次 `f{char}` 所查找的字符（参见技巧50）。

当不小心做过头时，知道怎么回退会很有帮助。表1-1总结了 Vim 中可重复执行的命令，以及相应的回退方式。在多数场景中，撤销

（undo）都是我们想要使用的命令，难怪我键盘上的 u 键磨损得这么厉害！

表1-1 可重复的操作及如何回退

| 目的 | 操作 | 重复 | 回退 |
|--------------|-----------------------|----|----|
| 做出一个修改 | {edit} | . | u |
| 在行内查找下一指定字符 | f{char} /t{char} | ; | , |
| 在行内查找上一指定字符 | F{char} /T{char} | ; | , |
| 在文档中查找下一处匹配项 | /pattern | n | N |
| 在文档中查找上一处匹配项 | ?pattern | n | N |
| 执行替换 | :s/target/replacement | & | u |
| 执行一系列修改 | qx{changes}q | @x | u |

技巧5 查找并手动替换

Vim提供了一个 :substitute 命令专门用于查找替换任务，不过用上面介绍的技术，也可以手动修改第一处地方，然后再一个个地查找替换其他匹配项。 . 命令可以把我们从繁重的工作中解放出来，而即将登场的另一个有用的单键命令，则能够让我们方便地在匹配项间跳转。

在下面这段文本中，每一行都出现了单词“content”。

the_vim_way/1_copy_content.txt

```
...We're waiting for content before the site can go live...  
...If you are content with this, let's go ahead with it...  
...We'll launch as soon as we have the content...
```

假设有想用单词“copy”（意义同“copywriting”）来替代“content”。也许你会想，这太简单了，只要用替换命令就行了，像下面这样：


⇒:%s/content/copy/g

但是，且慢！如果我们运行上面这条命令，就会出现“If you are ‘copy’ with this,”这样的句子，这很荒唐！

之所以会有这种问题，是因为“content”一词有两种含义，一个是“copy”的同义词（发音为'kon'tent），另一个是“happy”的同义词（发音为kən'tent）。用专业的话说，我们是在处理拼写相同，但含义和发音都不同的词。不过这不是我想说的重点，重点是我们一定要小心每一步操作。

我们不能想当然地用“copy”替换每一个“content”，而是要时刻留神，对每个地方都要问“这里要修改吗？”，然后回答“修改”或者“不改”。substitute 命令也能胜任这项工作，我们将在技巧90中学到该怎么做。不过现在，我们将寻求符合本章主题的另一解决办法。

偷懒的办法：无需输入就可以进行查找

现在你可能已经猜到了，. 命令是我最喜爱的Vim单键命令，而排在第二位的是 / 命令，此命令可以查找当前光标下的单词（参见 `:h`  ）。

我们可以调出查找提示符，并输入完整的单词来查找“content”。

⇒ /content

或者，可以简单地把光标移到这个单词上，然后按 * 键。以下面的操作为例。

| 按键操作 | 缓冲区内容 |
|--------------|--|
| {start} | ...We're waiting for content before the site can go live... |
| | ...If you are content with this, let's go ahead with it... |
| | ...We'll launch as soon as we have the content... |
| * | ...We're waiting for content before the site can go live... |
| | ...If you are content with this, let's go ahead with it... |
| | ...We'll launch as soon as we have the content ... |
| cw copy<Esc> | ...We're waiting for content before the site can go live... |
| | ...If you are content with this, let's go ahead with it... |
| | ...We'll launch as soon as we have the copy ... |
| n | ...We're waiting for content before the site can go live... |
| | ...If you are content with this, let's go ahead with it... |
| | ...We'll launch as soon as we have the copy... |

| 按键操作 | 缓冲区内容 |
|------|---|
| . | ...We're waiting for copy before the site can go live... |
| | ...If you are content with this, let's go ahead with it... |
| | ...We'll launch as soon as we have the copy... |

刚开始，把光标移到单词“**content**”上，然后使用 `*` 命令对它进行查找，你也可以自己试一下。这会产生两个结果：一是光标跳到下一个匹配项上，二是所有出现这个词的地方都被高亮显示出来。如果你并没有看到高亮，试着运行一下 `:set hls`。要了解更多这方面的内容，请参见技巧81。

执行过一次查找“**content**”的命令后，现在只需按 `n` 键就可以跳到下一个匹配项。在本例中，按 `*nn` 会遍历完所有的匹配项，从而跳回到本次查找的起点。

使修改可重复

当光标位于“**content**”的开头时，就可以着手修改它。这包括两步操作：首先要删除单词“**content**”，然后输入替代的单词。`cw` 命令会删除从光标位置到单词结尾间的字符，并进入插入模式，接下来就可以输入单词“**copy**”了。Vim会把我们离开插入模式之前的全部按键操作都记录下来，因此整个 `cw copy<Esc>` 会被当成一个修改。也就是说，执行 `.` 命令会删除从光标到当前单词结尾间的字符，并把它修改为“**copy**”。

合而为一

万事俱备！每次按 `n` 键时，光标就会跳到下一个“**content**”单词所在之处，而按 `.` 键时，它就会把光标下的单词改为“**copy**”。

如果想替换所有地方，就可以不加思考地一直按 `n.n.n.` 以完成所有的修改（但是，这种情况下也可以用 `:%s/content/copy/g` 命令）。然而，由于我们需要留意不符合要求的匹配项，所以在按了 `n` 之后，要审视一下当前的匹配项，然后决定是否把它改为“`copy`”。如果需要修改的话，就按 `.` 命令，反之则不用。无论决定是什么，都可以再次按 `n` 移到下一个地方，如此循环往复，直到完成全部的修改。

技巧6 认识 `.` 范式

到目前为止，我们介绍了3个简单的编辑任务。尽管每个问题都不一样，不过我们都找到了用 `.` 命令解决该问题的方法。在本节，我们将比较这些方案，并找出它们共有的模式——一个我称之为“`.` 范式”的最佳编辑模式。

回顾前面3个 `.` 命令编辑任务

在技巧2中，我们想在一系列行的结尾添加分号。我们先用 `A ;` `<Esc>` 修改了第一行，做完这步准备后，就可以使用 `.` 命令对后续行重复此修改。我们使用了 `j` 命令在行间移动，要完成剩余的修改，只需简单地按足够多次 `j.` 就可以了。

在技巧3中，我们想为每个 `+` 号的前后各添加一个空格。先用 `f+` 命令跳到目标字符上，然后用 `s` 命令把一个字符替换成3个，做完这步准备后，就可以按若干次 `;` `.` 完成此任务。

在技巧5中，我们想把每处出现单词“`content`”的地方都替换成“`copy`”。使用 `*` 命令来查找目标单词，然后用 `cw` 命令修改第一处地方。做完这步准备后，就可以用 `n` 键跳到下一匹配项，然后用 `.` 键做相同的修改。要完成这项任务，只需简单地按足够多次 `n.` 就行了。

理想模式：用一键移动，另一键执行

所有这些例子都利用 `.` 命令重复上次的修改，不过这不是它们唯一的共同点，另外的共同点是它们都只需要按一次键就能把光标移到下一个目标上。

用一次按键移动，另一次按键执行，再没有比这更好的了，不是吗？这就是我们的理想解决方案。我们将会一次又一次地看到这一编辑模式，所以为了方便起见，把它叫做“`. 范式`”。

第一部分 模式

Vim提供一个区分模式的用户界面，就是说在Vim中按键盘上的任意键所产生的结果可能会不一样，而这取决于当前处于哪种模式（mode）。知道当前正处于哪种模式，以及如何在各模式间切换，是极其重要的。在本书的这一部分，我们将学习每种模式的工作方式及其用途。

第2章 普通模式

普通模式是Vim的自然放松状态，如果本章看起来出奇的短，那是因为几乎整本书都在讲如何利用普通模式，而本章只涉及其中的一些核心概念以及通用技巧。

其他文本编辑器大部分时间都处于类似Vim插入模式的状态中，因此对Vim新手来说，把普通模式（normal mode）当成默认状态看起来很奇怪。在技巧7中，我们将以一个画家的工作区作为类比，来解释其原因。

许多普通模式命令可以在执行时指定执行的次数，这样它们就可以被执行多次。在技巧10中，我们将结识一对用于加减数值的命令，并且会看到这两条命令如何与次数结合在一起，进行简单的算术运算。

指定执行的次数可以减少按键个数，但并不是说一定要为此目的而这样做。我们将会看到一些例子，在这些例子中，简单地重复执行一条命令，要比花时间去计算想要执行多少次更好。

普通模式命令的强大，很大程度上源于它可以把操作符与动作命令结合在一起。在本章的最后，我们将看到这种结合达到的效果。

技巧7 停顿时请移开画笔

对于不习惯Vim的人来说，普通模式看上去是一种奇怪的缺省状态，但有经验的Vim用户却很难想象还有其他任何方式。本节使用了一个比喻来说明为什么Vim要采用这种方式。

你估计画家会花费多少时间用画笔在画布上作画？毫无疑问，这因人而异，但是，如果这占了画家全部工作时间的一半还要多的话，我会觉得非常诧异。

想一下除了画画外，画家还要做些什么事情。他们要研究主题，调整光线，把颜料混合成新的色彩，而且在把颜料往画布上画时，谁说他们必须要用画笔？画家也许会换用刻刀来实现不同的质地，或是用棉签来对已经画好的地方进行润色。

画家在休息时不会把画笔放在画布上。对Vim而言也是这样，普通模式就是Vim的自然放松状态，其名字已经寓示了这一点。

就像画家只花一小部分时间涂色一样，程序员也只花一小部分时间编写代码。绝大多数时间用来思考、阅读，以及在代码中穿梭浏览，而且当确实需要修改时，谁说一定要切换到插入模式才行？我们可以重新调整已有代码的格式，复制它们，移动其位置，或是删除它们。在普通模式中，我们有众多的工具可以利用。

技巧8 把撤销单元切成块

在其他编辑器中，输入一些词后使用撤销命令，可能会撤销最后输入的词或字符，然而在Vim中，我们自己可以控制撤销的粒度。

`u` 键会触发撤销命令，它会撤销最新的修改。一次修改可以是改变文档内文本的任意操作，其中包括在普通模式、可视模式以及命令行模式中所触发的命令，而且一次修改也包括了在插入模式中输入（或删除）的文本，因此我们也可以说，`i{insert some text}<Esc>` 是一次修改。

在不区分模式的文本编辑器中，输入一些单词后使用撤销命令，有两种可能。一种是它可能会撤销最后输入的字符；另一种做得更好

点，它可能会把字符分成块，使每次撤销操作删除一个单词而不是一个字符。

在Vim中，我们自己可以控制撤销命令的粒度。从进入插入模式开始，直到返回普通模式为止，在此期间输入或删除的任何内容都被当成一次修改。因此，只要控制好对 <Esc> 键的使用，就可使撤销命令作用于单词、句子或段落。

那么，应该多久离开一次插入模式呢？这是个人喜好的问题，不过我喜欢让每个“可撤销块”对应一次思考过程。在写这段文字时（当然是在Vim中写的），我经常在每句话的结尾停顿一下，想一想接下来该写什么。不管停顿的时间有多短，每次停顿都是一个自然的中断点，提示我该退出插入模式了。当我准备好继续写时，按 A 命令就可以回到原来的地方继续写作。

如果我认为已经走错了方向，就会切换到普通模式，然后按 u 撤销。每次做撤销时，文字都按我最初书写时的思路，被切分成条理清晰的块，也就是说我可以很容易地试着写一两句话，如果感到不合适的话，随后按一两下键就可以将其舍弃。

当处于插入模式时，如果光标位于行尾的话，另起一行最快的方式是按 <CR>。不过有时我更喜欢按 <Esc>O，这是因为我有预感，也许在撤销时我想拥有更细的粒度。如果听起来这不太好理解，不必担心，当你对Vim越来越熟悉时，就会感到切换模式越来越轻松。

一般来讲，如果你停顿的时间长到足以问“我应该退出插入模式吗？”这个问题，就退出吧。

在插入模式中移动光标会重置修改状态

当我提到撤销命令会回退从进入插入模式到退出此模式期间输入（或删除）的全部字符时，我略过了一个小细节。如果在插入模式中使用了 <Up>、<Down>、<Left> 或 <Right> 这些光标键，将会产生一个新的撤销块。你可以把这想象为先切换回普通模式，然后用 h、j、k 或 l 命令对光标进行了移动，唯一

区别是我们并没有退出插入模式。这也会对 . 命令的操作产生影响。

技巧9 构造可重复的修改

Vim对重复操作进行了优化，要利用这一点，必须考虑该如何构造修改。

在Vim中，要完成一件事，总是有不止一种方式。在评估哪种方式最好时，最显而易见的指标是效率，即哪种手段需要的按键次数最少（又名VimGolf^[1]）。然而，在平局时该如何选择获胜者呢？

在下例中，假设光标位于行尾处的字符“h”上，而我们想要删除单词“nigh”：

normal_mode/the_end.txt

```
The end is nigh
```

反向删除

因为光标已经位于单词末尾，所以可以先反向删除该词。

| 按键操作 | 缓冲区内容 |
|---------|-----------------|
| {start} | The end is nigh |
| db | The end is h |

| 按键操作 | 缓冲区内容 |
|----------|---------------------|
| x | The end is █ |

按 **db** 命令删除从光标起始位置到单词开头的内容，但会原封未动地留下最后一个字符“**h**”，再按一下 **x** 键就可以删除这个捣乱的字符。这样，整个操作的 Vim 高尔夫得分是3分。

正向删除

这一次，让我们尝试一下正向删除。

| 按键操作 | 缓冲区内容 |
|----------------|--------------------------|
| {start} | The end is nigh h |
| b | The end is n igh |
| dw | The end is █ |

先用 **b** 命令把光标移到单词的开头，移动好后，就可以用一个 **dw** 命令删掉整个单词。这一次的Vim高尔夫得分也是3分。

删除整个单词

到目前为止，已有的两种方式都要先做某种准备工作或清理工作。另外，也可以使用更为精准的 **aw** 文本对象（text object），而不是用动作命令（参见 **:h aw** ⓘ）。

| 按键操作 | 缓冲区内容 |
|---------|-----------------|
| {start} | The end is nigh |
| daw | The end is |

可以把 daw 命令解读为“delete a word”，这样比较容易记忆。在技巧52和技巧53中将介绍更多关于文本对象的细节。

决胜局：哪种方式最具重复性？

我们尝试了3种不同的方式来删除一个词：dbx、bdw 以及 daw。每种情况的Vim高尔夫得分都是3分。那么要怎么回答这个问题：“哪种方式最好？”

还记得吗，Vim对重复操作进行了优化。让我们再回顾一下这3种方式，这一次我们跟着用一次. 命令，看看会发生什么。我建议你自己也亲自试一下。

反向删除方案包含两步操作：db 命令删除至单词的开头，而后x 命令删除一个字符。如果我们跟着执行一次. 命令，它会重复删除一个字符（. == x）。我不觉得这有什么价值。

正向删除方案也包含两步。这一次，b 只是一次普通的移动，而dw 完成修改。此时用. 命令会重复 dw，删除从光标位置到下个单词开头的内容。不过因为我们刚好已经在行尾了，并没有“下一个单词”，所以在这个场景里. 命令没什么用。不过，至少它代表了一个更长点的操作（. == dw）。

最后的方案只调用一个操作：daw。这个操作不仅仅删除了该单词，它还会删除一个空格，因此光标最终会停在单词“is”的最后一个字符上。如果此时使用. 命令，它会重复上次删除单词的命令。这一次，. 命令会做真正有用的事情（. == daw）。

结论

`daw` 可以发挥 `.` 命令的最大威力，因此我宣布它是本轮的获胜者。

要想充分利用 `.` 命令，事先常常需要进行一番周详的考虑。如果你发现自己要在几个地方做同样的小修改，就可以尝试构造你的修改，让它们能够被 `.` 命令重复执行。要识别出这类机会需要进行一定的实践，不过一旦养成了使修改可重复的习惯，你就会从 Vim 这里得到“奖赏”。

有时，我并没有看到用 `.` 命令的机会，然而在做完一次修改后，我发现要做另一次同样的操作，这时候，我脑海里会浮现出 `.` 命令，而它也已经准备好为我效力了。每当遇到这种情况时，我都会开心地笑起来。

技巧10 用次数做简单的算术运算

大多数普通模式命令可以在执行时指定次数，可以利用这个功能来做简单的算术运算。

很多普通模式命令都可以带一个次数前缀，这样 Vim 就会尝试把该命令执行指定的次数，而不是只执行一次（参见 `:h count` ⓘ）。

`<C-a>` 和 `<C-x>` 命令分别对数字执行加和减操作。在不带次数执行时，它们会逐个加减，但如果带一个次数前缀，那么可以用它们加减任意整数。例如，如果把光标移到字符 5 上，执行 `10<C-a>` 就会把它变成 15。

但是如果光标不在数字上会发生什么？文档里说，`<C-a>` 命令会“把当前光标之上或之后的数值加上 `[count]`”（参见 `:h ctrl-a` ⓘ）。因此，如果光标不在数字上，那么 `<C-a>` 命令将在当前行正向查

找一个数字，如果找到了，它就径直跳到那里。我们可以利用这一点简化操作。

下面是一段CSS片段。

normal_mode/sprite.css

```
.blog, .news { background-image: url
(/sprite.png
); }
.blog { background-position: 0px 0px }
```

我们要复制最后一行并且对其做两个小改动，即用“news”替换单词“blog”，以及把“0px”改为“-180px”。可以运行 `yyp` 来复制此行，然后用 `cw` 来修改第一个单词。但该怎么处理那个数值呢？

一种做法是用 `f0` 跳到此数字，然后进入插入模式手动修改它的值，即 `i-18<Esc>`。不过，运行 `180<C-x>` 则要快得多。由于光标不在要操作的数字上，所以该命令会正向跳到所找到的第一个数字上，从而省去了手动移光标的步骤。让我们看看整个操作过程。

| 按键操作 | 缓冲区内容 |
|---------|--|
| {start} | .blog, .news { background-image: url(/sprite.png); } |
| | . blog { background-position: 0px 0px } |
| yyp | .blog, .news { background-image: url(/sprite.png); } |
| | .blog { background-position: 0px 0px } |
| | . blog { background-position: 0px 0px } |

| 按键操作 | 缓冲区内容 |
|----------------------|--|
| cW .news<Esc> | .blog, .news { background-image: url(/sprite.png); } |
| | .blog { background-position: 0px 0px } |
| | .news { background-position: 0px 0px } |
| 180 <C-x> | .blog, .news { background-image: url(/sprite.png); } |
| | .blog { background-position: 0px 0px } |
| | .news { background-position: -180 px 0px } |

在本例中，只复制了一行并做出改动。但是，假设要复制10份，并对后续数字依次减180。如果要切换到插入模式去修改每个数字，每次都得输入不同的内容（-180，然后-360，以此类推）。但是如果用180<C-x>命令的话，对后续行也可以采用相同的操作过程。甚至还可以把这组按键操作录制成一个宏（参见第11章），然后根据需要执行多次。

数字的格式

007的后面是什么？不，这不是詹姆斯·邦德的恶作剧，我是在问：如果对007加1，你觉得会得到什么结果。

如果你的答案是008，那么当你尝试对任意以0开头的数字使用<C-a>命令时，也许会感到诧异。像在某些编程语言中的约定一样，Vim把以0开头的数字解释为八进制值，而不是十进

制。在八进制体系中， $007 + 001 = 010$ ，看起来像是十进制中的10，但实际上它是八进制中的8，糊涂了吗？

如果你经常使用八进制，Vim的缺省行为或许会适合你。如果不是这样，那么你可能想把下面这行加入你的vimrc里：

```
set nrformats=
```

这会让Vim把所有数字都当成十进制，不管它们是不是以0开头的。

技巧11 能够重复，就别用次数

在处理某些特定工作时，使用次数可以使按键次数变得最少，不过并不是非得这样不可。我们需要认真考虑次数与重复各自的优缺点。

假设在缓冲区里有如下文字。

```
Delete m  
ore than one word
```

想把这段文字改为“Delete one word”，也就是说，要像这段文字里所讲的那样删除两个单词。

有几种方式可以达到这一目的，d2w 和 2dw 都可以。使用 d2w，先调用删除命令，然后以 2w 作为动作命令，可以把它解读为“删除两个单词”；然而 2dw 做的相反，这一次，次数作用于删除命令，而

动作命令只跨越一个单词，可以把这解读为“做两次删除单词的操作”。抛开语义不讲，无论哪种方法，结果都是相同的。

现在，让我们考虑另外一种方式，即`dw.`。这可以解读为“删除一个单词，然后重复上次的操作”。

概括一下，我们的3种选择 `d2w`、`2dw` 或者 `dw.` 都是3次按键，不过哪一种最好呢？

根据我们的讨论，`d2w` 和 `2dw` 是相同的，在执行完两者中的任何一个后，可以按 `u` 键撤销，这样两个被删除的单词又会回来。或者，我们不是用撤销，而是用 `.` 命令重复执行它，这就会删除后面的两个单词。

对于 `dw.` 的情形，按 `u` 或 `.` 的结果会有细微的差别。这里的修改是 `dw`，即删除一个单词。因此，如果想恢复这两个被删除的单词，必须撤销两次，按 `uu`（或者，如果你愿意，也可以按 `2u`）。按 `.` 则只删除后面的一个单词，而不是两个。

现在假设我们原本是想删除3个单词，而不是2个。由于判断出了点差错，我们执行了 `d2w` 而不是 `d3w`，那接下来怎么做？我们不能使用 `.` 命令，因为那会总共删除4个单词。因此，我们或是先撤销而后修正次数（`ud3w`），或是继续删除下一个单词（`dw`）。

现在考虑另一种方案，如果我们在第一处地方用的是 `dw.` 命令，那么只要再多重复一次 `.` 命令就行了。因为我们最初的修改只是简单的 `dw`，因此`u`命令和`.`命令都具有更细的粒度，每次只作用于一个单词。

现在假设我们想删除7个单词，我们可以运行 `d7w`，或是 `dw.....`（即 `dw` 后面跟6次 `.` 命令）。计算一下按键的次数，哪个命令胜出是很显而易见的。不过你真地确信自己数对了次数吗？

计算次数很是讨厌，因此我宁愿按6次 `.` 命令，也不愿意只为减少按键的次数，而浪费同样的时间去统计次数。如果我多按了一次 `.` 命令怎么办？没关系，只要按一次 `u` 键就可以回退回来。

还记得吗，我们的口诀是（参见技巧4）：执行、重复、回退。这里就是在把它付诸行动。

只在必要时使用次数

假设我们想把文字“I have a couple of questions”改为“I have some more questions”，可以用下面的方式做。

| 按键操作 | 缓冲区内容 |
|--------------------|-------------------------------|
| {start} | I have a couple of questions. |
| c3w some more<Esc> | I have some more questions. |

在此场景中，使用 `.` 命令的意义不大，我们可以删除一个单词，然后再用 `.` 命令删除另一个，但随后我们还得切换到插入模式（例如，使用 `i` 或 `cw`）。对我来说这么做很不顺手，我反而更愿意用次数。

使用次数的另一个好处是：它保留了一个干净、连贯的撤销历史记录。完成这次修改后，按一下 `u` 键就可以撤销整个修改，这和技巧8中的讨论是一致的。

对于是用次数风格（`d5w`）还是用重复风格（`dw....`）也有同样的争论，因此我的偏好看起来似乎不太一致。对此，你要总结自己的观点，这取决于你怎么看保留干净撤销历史记录的价值，以及你是否觉得用次数令人生厌。

技巧12 双剑合璧，天下无敌

Vim的强大很大程度上源自操作符与动作命令相结合。在本节，我们将看到它是如何工作的，并考虑其寓意。

操作符 + 动作命令 = 操作

`d{motion}` 命令可以对一个字符 (`dl`)、一个完整单词 (`daw`) 或一整个段落 (`dap`) 进行操作，它作用的范围由动作命令决定。`c{motion}`、`y{motion}` 以及其他一些命令也类似，它们被统称为操作符 (operator)。可以用 `:h operator` ❶ 来查阅完整的列表，表2-1总结了一些比较常见的操作符。

`g~`、`gu` 和 `gU` 命令要用两次按键来调用，我们可以把上述命令中的 `g` 当作一个前缀字符，用以改变其后面的按键行为，进一步的讨论请参见本技巧最后的“结识操作符待决模式”部分。

操作符与动作命令的结合形成了一种语法。这种语法的第一条规则很简单，即一个操作由一个操作符，后面跟一个动作命令组成。学习新的动作命令及操作符，就像是在学习Vim的词汇一样。如果掌握了这一简单的语法规则，在词汇量增长时，就能表达更多的想法。

假如我们已经知道如何用 `daw` 删除一个单词，然后又学到 `gU` 命令（参见 `:h gU` ❶）。它也是个操作符，所以可以用 `gUaw` 把当前单词转换成大写形式。如果我们的词汇进一步扩充，学会了作用于段落的 `ap` 动作命令，就会发现我们可以进行两个新的操作：用 `dap` 删除整个段落，或者用 `gUap` 把整段文字转换为大写。

Vim的语法只有一条额外规则，即当一个操作符命令被连续调用两次时，它会作用于当前行。所以 `dd` 删除当前行，而 `>>` 缩进当前行。`gU` 命令是一种特殊情况，我们既可以用 `gUgU`，也可以用简化版的 `gUU` 来使它作用于当前行。

表2-1 Vim的操作符命令

| 命令 | 用途 |
|----------|----|
| c | 修改 |

| 命令 | 用途 |
|-------------|-------------------------|
| d | 删除 |
| y | 复制到寄存器 |
| g~ | 反转大小写 |
| gu | 转换为小写 |
| gU | 转换为大写 |
| > | 增加缩进 |
| < | 减小缩进 |
| = | 自动缩进 |
| ! | 使用外部程序过滤 {motion} 所跨越的行 |

扩展命令组合的威力

使用Vim缺省的操作符和动作命令，我们能够执行的操作的数目是巨大的，然而，我们还可以通过自定义动作命令及操作符来进一步扩充其数目。让我们想想这寓示着什么。

自定义操作符与已有动作命令协同工作

随同Vim发布的标准操作符集合相对比较少，但可以定义新的操作符。Tim Pope的commentary.vim插件提供了一个很好的例子 [\[2\]](#)，此

插件为Vim支持的编程语言增添了注释及取消注释的命令。

注释命令以 `gc{motion}` 触发，它会切换指定行的注释状态。它是一个操作符命令，因此可以把它和所有动作命令结合在一起。`gcap` 将切换当前段落的注释状态，`gcG` 会把从当前行到文件结尾间的所有内容注释掉，`gcc` 则注释当前行。

如果你对如何创建自定义操作符感到好奇，可以先阅读一下文档 `:h:map-operator` ①。

自定义动作命令与已有操作符协同工作

Vim缺省的动作命令集已经相当全面了，但是我们还是可以定义新的动作命令及文本对象来进一步增强它。

Kana Natsuno的`textobj-entire`插件是一个很好的例子 [3]，它为Vim增加了两种新的文本对象 `ie` 和 `ae`，它们作用于整个文件。

如果想用 `=` 命令自动缩进整个文件，可以执行 `gg=G`（就是说，先用 `gg` 跳到文件开头，然后用 `=G` 自动缩进从光标位置到文件结尾的所有内容）。但是如果安装了`textobj-entire`插件的话，简单地执行 `=ae` 就可以了。运行这条命令时光标在哪儿并不重要，因为它总是作用于整个文件。

注意：

如果同时安装了`commentary`和`textobj-entire`插件，就可以把它们放在一起使用。例如，执行 `gcae` 会切换整个文件的注释状态。

如果你对如何创建自定义动作命令感到好奇，可以由阅读 `:h omap-info` ① 开始。

结识操作符待决模式

普通、插入及可视模式很容易辨识，但是Vim还有另外一些很容易被忽视的模式，操作符待决模式（`operator-pending`）

mode) 就是一个例子。每天我们无数次地使用它，但通常它只持续不到一秒时间。举个例子，在执行命令 `dw` 时，就会激活该模式。这一模式只在按 `d` 及 `w` 键之间的短暂时间间隔内存在，一眨眼工夫就不见了。

如果把Vim想象成有限状态机，那么操作符待决模式就是一个只接受动作命令的状态。这个状态在调用操作符时被激活，然后什么也不做，直到我们提供了一个动作命令，完成整个操作。当操作符待决模式被激活时，我们可以像平常一样按 `<Esc>` 中止该操作，返回到普通模式。

很多命令都由两个或更多的按键来调用（查阅 `:h g`^❶、`:h z`^❶、`:h ctrl-w`^❶，或者 `:h [`^❶，可以看到一些例子），但在多数情况下，头一个按键只是第二个按键的前缀。这些命令不会激活操作符待决模式，相反，可以把它们当成命名空间（`namespace`），用来扩充可用命令的数目。只有操作符才会激活操作符待决模式。

你也许想知道，为什么要有一个完整的模式，专门用于操作符和动作命令之间的短暂瞬间，而命名空间命令则仅仅是普通模式的一个扩充？好问题！这是因为我们能够创建自定义映射项来激活或终结操作符待决模式。换句话说，它允许我们创建自定义的操作符及动作命令，从而让我们可以扩充Vim的词汇。

[1] <http://vimgolf.com/>

[2] <https://github.com/tpope/vim-commentary>

[3] <https://github.com/kana/vim-textobj-entire>

第3章 插入模式

大部分的Vim命令都在非插入模式中执行，不过有些功能在插入模式中会更好实现些，我们将在本章深入研究这些命令。尽管删除、复制以及粘贴命令都是在普通模式中执行的，不过我们将会看到一种方便快捷的方式，让我们无需离开插入模式就能粘贴寄存器中的文本。另外我们也会学习Vim提供的两种简单方式，用来插入键盘上不存在的非常用字符。

替换模式是插入模式的一种特例，它会替换文档中已有的字符。我们将学习如何使用它，并了解在哪些场景下它能够大显身手。我们也会结识插入-普通模式，它是一个子模式，可以让我们执行一个普通模式命令，之后马上又回到插入模式。

自动补全是插入模式中才能使用的高级功能，我们将在第19章对其进行深入的研究。

技巧13 在插入模式中可即时更正错误

如果在插入模式下撰写文本时出了错，可以立刻对它进行更正，而无需切换模式。要迅速更正错误，除了用退格键外，还可以用插入模式中的其他一些命令。

盲打并不仅仅指输入时不看键盘，还意味着输入时要凭感觉。当盲打的人输入错误时，在眼睛看到屏幕上的错误之前，他们就已经察觉到了，他们可以用手指感知到次序颠倒这类的错误。

在输入错误时，可以用退格键删除错误的文本，然后再输入正确的内容。如果出错的地方靠近单词结尾，这或许是最快的修正方式。但是，如果出错的位置在单词开头呢？

专业打字员会建议先删除整个单词，然后再重新输入一遍。如果你能以每分钟超过60个单词的速度输入，那么重新输入一个词只需要1秒钟的时间。即便你打不了这么快，最好也采用这种方式。我以前总是输错某些特定的词，但自从采纳这一建议后，我就更清楚地意识到哪些词会让我犯错，因此现在犯的错也少了很多。

另外，也可以切换到普通模式，然后跳到这个词的开头并更正错误，再按 `A` 返回刚才的位置。不过完成这一套动作要花的时间可能不止1秒钟，并且它也无助于提高你的盲打技巧。虽然说我们可以切换模式，不过这并不意味着一定就得切换。

在插入模式下，退格键的作用如你所愿，它删除光标前的字符。另外，还可以用下面这些组合键。

| 按键操作 | 用途 |
|--------------------------|---------------|
| <code><C-h></code> | 删除前一个字符（同退格键） |
| <code><C-w></code> | 删除前一个单词 |
| <code><C-u></code> | 删至行首 |

这些命令不是插入模式独有的，甚至也不是Vim独有的，在Vim的命令行模式中，以及在 `bash shell` 中，也可以使用它们。

技巧14 返回普通模式

插入模式只专注于做一件事，那就是输入文字，而普通模式却是我们大部分时间所使用的模式（顾名思义），因此能快速在这两种模式间切换是很重要的。本节将介绍一些可以减少模式切换所带来的损耗的技巧。

切换回普通模式的经典方式是使用 `<Esc>` 键，但在许多键盘上这个键的距离似乎有点远。作为替代，也可以用 `<C-[>`，它的效果与 `<Esc>` 完全相同（参见 `:h i_CTRL-[` ❶）。

| 按键操作 | 用途 |
|--------------------------|------------|
| <code><Esc></code> | 切换到普通模式 |
| <code><C-[></code> | 切换到普通模式 |
| <code><C-o></code> | 切换到插入-普通模式 |

Vim新手经常会被不断地切换模式而搞得疲倦不堪，不过练习过一段时间以后，就会渐渐感觉到得心应手了。不过，Vim区分模式的特点在下面这种特定场景中却显得有点烦琐：处于插入模式时，想运行一个普通模式命令，然后马上回到原来的位置继续输入。Vim为此提供了一种巧妙的方法，以减少模式切换所带来的不畅，这就是插入-普通模式。

结识插入-普通模式

插入-普通模式是普通模式的一个特例，它能让我们执行一次普通模式命令。在此模式中，可以执行一个普通模式命令，执行完后，马上又返回到插入模式。要从插入模式切换到插入-普通模式，可以按 `<C-o>`（参见 `:h i_CTRL-``O` ❶）。

在当前行正好处于窗口顶部或底部时，有时我会滚动一下屏幕，以便看到更多的上下文。用 `zz` 命令可以重绘屏幕，并把当前行显示在窗口正中，这样就能够阅读当前行之上及之下的半屏内容。我常常键入 `<C-o>zz`，在插入-普通模式中触发这条命令。此操作完成后就会直接回到插入模式，因此可以不受中断地继续打字。

技巧15 不离开插入模式，粘贴寄存器中的文本

Vim的复制和粘贴操作一般都在普通模式中执行，不过有时我们也许想不离开插入模式，就能往文档里粘贴文本。

下面是一段尚未完成的文本。

insert_mode/practical-vim.txt

```
Practical Vim, by Drew Neil  
Read Drew Neil's
```

重新映射大小写转换键（Caps Lock）

对Vim用户而言，大小写转换键是一个威胁。如果大小写转换键处于大写模式，而你尝试用k或j去移动光标，那么你触发的将会是K和J命令。简单地讲，K命令用于查看处于光标之下的那个单词的手册页（参见：h K` ❶），J命令则用来把当前行和下一行连接在一起（参见：h J ❶）。也就是说，如果你不小心切换到了大写模式，你将会惊讶地发现，缓冲区的内容怎么这么快就乱掉了！

很多Vim用户都会重新映射大小写转换键，把它当成另外一个键用，如<Esc>或<Ctrl>。在现代键盘上，<Esc>键很难够得到，而大小写转换键却很方便。把大小写转换键映射成<Esc>键可以省很多力气，尤其是Vim对<Esc>键用得这么频繁。不过我更喜欢把大小写转换键映射为<Ctrl>键。<C-[>的功用和<Esc>键相同，如果<Ctrl>键触手可及，那么这一组合键输入起来也会很容易。另外，不管是在Vim中还是在其他程序中，很多快捷键也都会用到<Ctrl>。

要重新映射大小写转换键，最简单的方法是在操作系统级别进行映射。不过对于OS X、Linux及Windows来说，其映射方法各不相同。因此，我不会在这儿重复每种系统的映射方法，而是建议你用Google搜索一下。注意，这一定制不仅会影响Vim，还会作用于整个系统。不过，如果你照我的建议做，你将会永远忘掉大小写转换键，我保证你不会怀念它。

我们想把本书的书名插到最后一行，以补全该行。由于书名在第一行的开头已经出现过了，所以将把它复制到一个寄存器中，然后在插入模式中把它添加到第二行结尾。

| 按键操作 | 缓冲区内容 |
|---------------------|---|
| y t, | P ractical Vim, by Drew Neil Read Drew Neil's |
| j A _ | Practical Vim, by Drew Neil Read Drew Neil's █ |
| <C-r>0 | Practical Vim, by Drew Neil Read Drew Neil's Practical Vim █ |
| .<Esc> | Practical Vim, by Drew Neil Read Drew Neil's Practical Vim. |

yt, 命令把“Practical Vim”复制到复制专用寄存器中（将在技巧50中结识 `t{char}` 动作命令），然后在插入模式中，按 **<C-r>0** 把刚才复制的文本粘贴到光标所在位置（将在第10章以大量的篇幅介绍寄存器以及复制操作）。

这个命令的一般格式是 `<C-r>{register}`，其中 `{register}` 是想要插入的寄存器的名字（参见 `:h i_CTRL-R` ①）。

对面向字符的寄存器使用 `<C-r>{register}` 命令

在插入模式中，可以用 `<C-r>{register}` 命令很方便地粘贴几个单词。可是如果寄存器中包含了大量的文本，你也许会发现屏幕的更新有些轻微的延时。这是因为Vim在插入寄存器内的文本时，其插入方式就如同这些文本是由键盘上一个个输进来的。因此，如果 `'textwidth'` 或者 `'autoindent'` 选项被激活了的话，最终就可能会出现不必要的换行或额外的缩进。

`<C-r><C-p>{register}` 命令则会更智能一些，它会按原义插入寄存器内的文本，并修正任何不必要的缩进（参见 `:h i_CTRL-R_CTRL-P` ①），不过这个命令有点不太好输入！因此，如果我想从一个寄存器里粘贴很多行文本的话，我更喜欢切换到普通模式，然后使用某个粘贴命令（参见技巧63）。

技巧16 随时随地做运算

表达式寄存器允许我们做一些运算，并把运算结果直接插入文档中。本节将看到一个运用此强大功能的实例。

大部分的Vim寄存器中保存的都是文本，要么是一个字符串，要么是若干行的文本。删除及复制命令允许我们把文本保存到寄存器中，粘贴命令则允许把寄存器中的内容插入文档里。

不过表达式寄存器则是个另类，它可以用来执行一段Vim脚本，并返回其结果。在本节，我们将把它当成计算器来用。传给它一个简单的算术表达式，如 `1 + 1`，它就会给出结果 `2`。对表达式寄存器所返回的文本，我们可以像用普通寄存器中的文本那样使用它。

可以用 = 符号指明使用表达式寄存器。在插入模式中，输入 <C-r>= 就可以访问这一寄存器。这条命令会在屏幕的下方显示一个提示符，可以在其后输入要执行的表达式。输入表达式后敲一下 <CR>，Vim就会把执行的结果插入文档的当前位置了。

假设刚输入完下列内容。

insert_mode/back-of-envelope.txt

```
6 chairs, each costing $35, totals $
```

我们想算一下总价，不过没必要找个信封在背面做演算，Vim可以帮我们做这件事，我们甚至连插入模式都不用退出。做法如下。

| 按键操作 | 缓冲区内容 |
|-----------------|--|
| A | 6 chairs, each costing \$35, totals \$█ |
| <C-r>= 6*35<CR> | 6 chairs, each costing \$35, totals \$210█ |

表达式寄存器远不止能做简单算术运算。我们将在技巧71中看到一些更高级的应用。

技巧17 用字符编码插入非常用字符

Vim可以用字符编码（Character Code）插入任意字符。使用此功能可以很方便地输入键盘上找不到的符号。

只要知道某个字符的编码，就可以让Vim插入该字符，我们可以用这种方式插入任意字符。要根据字符编码插入字符，只需在插入模式中输入 `<C-v>{code}` 即可，其中 {code} 是要插入字符的编码。

Vim接受的字符编码共包含3位数字。例如，假设想插入大写字母“A”，它的字符编码是65，因此需要输入 `<C-v>065` 。

然而，如果想插入一个编码超过3位数的字符该怎么办？例如，Unicode基本多文种平面（Unicode Basic Multilingual Plane）的地址空间最大会有65 535个字符。解决方法是用4位十六进制编码来输入这些字符，即输入 `<C-v>u{1234}`（注意数字前的u）。假设想插入字符编码为00bf的反转问号（“¿”），只需在插入模式中输入 `<C-v>u00bf` 即可。更多详细内容可参见 `:h i_CTRL-V_digit`` ①。

如果你想知道文档中任意字符的编码，只需把光标移到它上面并按 `ga` 命令，然后屏幕下方会显示出一条消息，分别以十进制和十六进制的形式显示出其字符编码（参见 `:h ga` ②）。当然，如果你想知道文档中不存在的字符的编码，该命令就无能为力了。在这种情况下，你或许得去查一下unicode表。

另外，如果 `<C-v>` 命令后面跟一个非数字键，它会插入这个按键本身代表的字符。例如，如果启用了 `'expandtab'` 选项，那么按 `<Tab>` 键将会插入空格而不是制表符。然而，按 `<C-v><Tab>` 则会一直插入制表符，不管 `'expandtab'` 选项激活与否。

表3-1对输入非常用字符的命令进行了总结。

表3-1 插入非常用字符

| 按键操作 | 用途 |
|---------------------------------|---------------|
| <code><C-v>{123}</code> | 以十进制字符编码插入字符 |
| <code><C-v>u{1234}</code> | 以十六进制字符编码插入字符 |

| 按键操作 | 用途 |
|--|-----------------------------|
| <code><C-v>{nondigit}</code> | 按原义插入非数字字符 |
| <code><C-k>{char1}{char2}</code> | 插入以二合字母{char1}{char2} 表示的字符 |

技巧18 用二合字母插入非常用字符

虽然Vim允许我们用数字编码插入任意字符，不过这既难记忆也难输入。我们也可以用二合字母（digraph）来插入非常用字符，成对的字符更容易记忆一些。

二合字母用起来很方便。在插入模式中，只需输入 `<C-k>{char1}{char2}` 即可。因此，如果想输入以二合字母 `?I` 表示的“¿”字符，可以简单地输入 `<C-k>?I`。

Vim在选择组成二合字母的两个字符时，尽量使之具有描述性，这样就更容易记住它们，甚至也能猜出其含义。例如，左右书名号《和》分别以二合字母`<<及>>`表示，普通分数（或常用分数） $\frac{1}{2}$ 、 $\frac{1}{4}$ 和 $\frac{3}{4}$ 则分别以二合字母`12`、`14`和`34`来表示。Vim的缺省二合字母集依从一定的惯例，`:h digraphs-default` ⓘ 文档对此进行了总结。

用命令 `:digraphs` ⓘ 可以查看可用的二合字母列表，不过该命令的输出不太好阅读。也可以用 `:h digraph-table` ⓘ 查看另一个更为有用的列表。

技巧19 用替换模式替换已有文本

在替换模式下输入会替换文档中的已有文本，除此之外，该模式与插入模式完全相同。

假设有如下一段文本。

insert_mode/replace.txt

```
Typing in Insert mode extends the line. But in Replace mode
the line length doesn't change.
```

我们想把这两个单独的句子合并成一句话，为此，需要把句号改成逗号，并将单词“**But**”中的“**B**”改为小写。下例展示了如何用替换模式完成这项工作。

| 按键操作 | 缓冲区内容 |
|--------------------------------------|---|
| {start} | T yping in Insert mode extends the line. But in Replace mode the line length doesn't change. |
| f. | Typing in Insert mode extends the line. But in Replace mode the line length doesn't change. |
| R <code>,_b<Esc></code> | Typing in Insert mode extends the line, b ut in Replace mode the line length doesn't change. |

用 **R** 命令可以由普通模式进入替换模式，然后就如例中所示，输入“`,_b`”替换原有的“`.`_`B`”字符。完成替换后，可以按 `<Esc>` 键返回普通模式。如果你的键盘上有 `<Insert>` 键，那么也可以用该键在插入模式和替换模式间切换，不过并非所有的键盘都有这个键。

用虚拟替换模式替换制表符

某些字符会使替换模式变得复杂化。以制表符为例，在文件中它以单个字符表示，但在屏幕上它却会占据若干列的宽度，此宽度由 `'tabstop'` 设置决定（参见 `:h 'tabstop'` ①）。如果把光标移到制表符上，然后进入替换模式，那么输入的下一个字符将会替换制表符。假设 `'tabstop'` 选项设置为8（这是缺省值），那么该操作的结果就是把8个字符替换成了一个字符，这将大幅缩短当前行的长度。

不过Vim还有另外一种替换模式，称为虚拟替换模式（**virtual replace mode**）。该模式可由 `gR` 命令触发，它会把制表符当成一组空格进行处理。假设把光标移到一个占屏幕8列宽的制表符上，然后切换到虚拟替换模式，在输入前7个字符时，每个字符都会被插入制表符之前；最后，当输入第8个字符时，该字符将会替换制表符。

在虚拟替换模式中，我们是按屏幕上实际显示的宽度来替换字符的，而不是按文件中所保存的字符进行替换。这会减少意外情况的发生，因此建议在可能的情况下尽量使用虚拟替换模式。

Vim也提供了单次版本的替换模式及虚拟替换模式。`r{char}` 和 `gr{char}` 命令允许覆盖一个字符，之后马上又回到普通模式（参见 `:h r` ①）。

第4章 可视模式

Vim的可视模式允许选中一块文本区域并在其上进行操作，从表面上看这应该很容易理解，因为大多数编辑软件都沿用此模式。然而，Vim的可视模式和其他软件的做法却截然不同，所以在本章的一开始，我们先深入了解可视模式（参见技巧20）。

Vim具有3种不同的可视模式，分别用于操作字符文本、行文本和块文本。我们将会探讨在这几种模式间切换的方法，并介绍一些更改选区边界的有用技巧（参见技巧21）。

也可以利用 `.` 命令来重复执行可视模式中的命令，然而只有在操作面向行的选区时，它才特别有用；而在操作面向字符的选区时，有时它无法达到我们的预期。我们将会看到，在这种情况下可能更适合用操作符命令。

列块可视模式非常特别，它允许在块状文本上进行操作。你会发现此功能有很多的用处，不过只着重介绍3个小技巧，展示其部分功能。

技巧20 深入了解可视模式

可视模式允许选中一个文本区域并在其上操作。尽管这看起来似乎很熟悉，但对于选中的文本，Vim的视角却有益于其他文本编辑器。

假设，我们暂时没有在Vim中工作，而是在网页里的文本框中输入了单词“March”，但发现应该输入的是“April”。因此，先用鼠标双击此单词，把它高亮选中后，按退格键删除它，然后再输入正确的月份。

可能你已经知道了，其实在本例中并没有必要按退格键。当单词“**March**”处于选中状态时，我们只需敲字母“A”，就会替换掉所选内容，清出地方来输入“**April**”的剩余部分。尽管节省的按键次数有限，但毕竟积少成多。

如果你期待Vim的可视模式也能以这种方式工作，那你就会觉得意外了。顾名思义，可视模式仅仅是另外一种模式，也就是说在此模式中，每个按键都完成一种不同的功能。

你已经熟识的很多普通模式命令，它们在可视模式中也能完成相同的功能。我们仍可以把 h、j、k 及 l 当成光标键使用；也可以用 f{char} 跳到当前行的某个字符上，然后用 ; 和 , 命令相应地正向或反向重复此跳转；甚至还可以用查找命令（以及 n / N 命令）跳转到匹配指定模式的地方。每次在可视模式中移动光标，都会改变高亮选区的边界。

某些可视模式命令执行的基本功能与普通模式相同，但操作上有些细微的变化。例如，在这两种模式中，c 命令的功能是一样的，都是删除指定的文本并切换到插入模式。不过，要指定其操作的范围，二者的方式却不甚相同。在普通模式中，先触发修改命令，然后使用动作命令指定其作用范围。如果你还记得技巧12中讲过的内容，就会知道这个命令被称为操作符命令。然而，在可视模式中，要先选中选区，然后再触发修改命令。这种次序颠倒的方式对所有的操作符命令都适用（参见表2-1 Vim的操作符命令）。对大多数人来说，可视模式的做法感觉起来更自然。

让我们回顾一下前面遇到的那个简单例子，即把单词“**March**”修改为“**April**”。这一次，假设我们不是在网页上的文本框里，而是回到了舒适的Vim中。我们先把光标移到单词“**March**”的某个位置，然后执行 viw 来高亮选择这个词。现在不能直接输入单词“**April**”，因为这会触发 A 命令并把文本“**pril**”添加到行尾。我们要换种做法，先用 c 命令修改所选内容，把这个单词删掉并进入插入模式，然后就可以输入完整的“**April**”了。这种做法和最初在网页中所用的方式类似，只不过用的是 c 键而不是退格键。

结识选择模式

在一个典型的文本编辑器环境中，当选中一段文本后，再输入任意可见字符时，这些选中的文本将会被删除。虽然Vim的可视模式未遵从此惯例，但是其选择模式（**Select Mode**）却按此方式工作。根据Vim的内置文档所述，选择模式“类似于Microsoft Windows的选择模式”（参见 `:h Select-mode` ①）。在此模式下，输入的可见字符会使选中的文本被删除，同时Vim会进入插入模式，并插入这个可见字符。

按 `<C-g>` 可以在可视模式及选择模式间切换。切换后看到的唯一不同是屏幕下方的提示信息会在“-- 可视 --”（--VISUAL --）及“--选择--”（--SELECT--）间转换。但是，如果在选择模式中输入任意可见字符，此字符会替换所选内容并切换到插入模式。当然，如果是在可视模式中，仍可以像往常一样用 `c` 键来修改所选内容。

如果你乐于接受 Vim 区分模式的特性，那么你应该很少会用到选择模式。这一模式的存在，只是为了迎合那些想让Vim更像其他文本编辑器的用户。我只知道有一处地方会用到选择模式。有一个模拟 TextMate 的 snippet 功能的插件，它会用选择模式来高亮当前的占位符。

技巧21 选择高亮选区

可视模式的3个子模式用于处理不同类型的文本。我们将在本节看到如何激活每种子模式，以及如何在它们之间切换。

Vim有3种可视模式。在面向字符的可视模式中，我们能够选择任意的字符范围，不论它是单个字符，还是位于一行内，或是跨若干行

的指定字符范围，都没问题。该模式适用于操作单词或短语。如果我们想对整行进行操作，可以改用面向行的可视模式。而面向列块的可视模式则允许对文档中的列块进行操作。列块可视模式非常特别，所以会在技巧24、技巧25和技巧26中花大量篇幅对其进行介绍。

激活可视模式

v 键是通往可视模式的大门。在普通模式下，按 v 可激活面向字符的可视模式，按 V（v和Shift键一起按）可激活面向行的可视模式，而按 <C-v>（v和Ctrl键一起按）则可激活面向列块的可视模式，请参见下表中的汇总。

| 命令 | 用途 |
|--------------------|-------------|
| v | 激活面向字符的可视模式 |
| V | 激活面向行的可视模式 |
| <C-v> | 激活面向列块的可视模式 |
| gv | 重选上次的高亮选区 |

gv 命令是个有用的快捷键，它用来重选上一次由可视模式选择的文本范围。不管上个选区是面向字符的、面向行的，还是面向列块的，gv 命令都能够正确地工作。不过如果上次的选区被删除了，它也许会工作得不太正常。

在可视模式间切换

可以在不同风格的可视模式间切换，方式与在普通模式下激活可视模式的方式相同。如果当前处于面向字符的可视模式，可以按 v 来切换到面向行的可视模式，或是用 <C-v> 来切换到面向列块的可视模式。然而，如果在面向字符的可视模式中再次按 v，就会回到普通模

式。所以，可以把 `v` 键当成在普通模式及面向字符的可视模式间转换的开关，`v` 及 `<C-v>` 键也一样可以在普通模式及其对应的可视模式间切换。当然了，你总是可以按 `<Esc>` 或 `<C-[>` 回到普通模式（就像退出插入模式那样）。下表总结了在可视模式间切换的命令。

| 按键操作 | 用途 |
|--|--|
| <code><Esc> / <C-[></code> | 回到普通模式 |
| <code>v / V / <C-v></code> | 切换到普通模式（在对应的面向字符可视模式、面向行的可视模式和面向列块的可视模式中使用时） |
| <code>V</code> | 切换到面向字符的可视模式 |
| <code>V</code> | 切换到面向行的可视模式 |
| <code><C-v></code> | 切换到面向列块的可视模式 |
| <code>O</code> | 切换高亮选区的活动端 |

切换选区的活动端

高亮选区的范围由其两个端点界定。其中一端固定，另一端可以随光标自由移动，可以用 `o` 键来切换其活动的端点。在定义选区时，如果定义到一半，才发现选区开始的位置不对，此时用这个键会很方便，不用退出可视模式再从头开始，只需按一下 `o`，然后重新调整选区的边界即可。下面的操作对此功能进行了演示。

| 按键操作 | 缓冲区内容 |
|------|-------|
|------|-------|

| 按键操作 | 缓冲区内容 |
|------------|-----------------------------------|
| {start} | Select from here to h ere. |
| vbb | Select from here to h ere. |
| o | Select from here to h ere. |
| e | Select from here to here . |

技巧22 重复执行面向行的可视命令

当使用 `.` 命令重复对高亮选区所做的修改时，此修改会重复作用于相同范围的文本。本节将修改一个面向行的高亮选区，然后使用 `.` 命令重复此修改。

在可视模式中执行完一条命令后，会返回到普通模式，并且在可视模式中选中的文本范围也不再高亮显示了。那么，如果想对相同范围的文本执行另外一条可视模式命令，该怎么办？

假设如下Python代码的缩进有些问题。

visual_mode/fibonacci-malformed.py

```
def
    fib(n):
        a, b = 0, 1
        while
```



```
    a < n:
print
    a,
a, b = b, a+b
fib(42)
```

这段代码的每级缩进使用4个空格，首先对Vim进行配置，使之符合此缩进风格。

准备工作

要想让 < 和 > 命令正常工作，需要把 'shiftwidth' 及 'softtabstop' 的值设为4，并启用 'expandtab' 选项。如果想了解这些配置是如何协同工作的，请查阅Vimcasts.org [\[1\]](#) 上的“Tabs and Spaces”主题。下面一行命令会完成上述设置。

⇒ :set shiftwidth=4 softtabstop=4 expandtab

先缩进一次，然后重复

在这段缩进错误的Python代码中，while关键字下面的两行应该多缩进两级。可以高亮选择这两行，然后用 > 命令对它进行缩进，以修正其缩进错误。但此操作只增加一级缩进就返回普通模式了。

要解决此问题，一个办法是使用 gv 命令重选相同的文本，然后再次调用缩进命令。然而，如果你已经对Vim解决问题的方式有所领悟的话，脑海里应该会响起警钟。

当需要执行重复操作时，. 命令是最佳的解决方案。与其手动重选相同范围的文本并执行相同的命令，倒不如直接在普通模式里按 . 键。下面是具体的操作。

| 按键操作 | 缓冲区内容 |
|------|-------|
|------|-------|

| 按键操作 | 缓冲区内容 |
|---------|--|
| {start} | <pre>def fib(n): a, b = 0, 1 while a < n: print a, a, b = b, a+b fib(42)</pre> |
| Vj | <pre>def fib(n): a, b = 0, 1 while a < n: print a, a , b = b, a+b fib(42)</pre> |
| >. | <pre>def fib(n): a, b = 0, 1 while a < n: print a, a, b = b, a+b fib(42)</pre> |

如果你善于计算的话，也许更乐意在可视模式中执行 2> 以便一步到位。不过我更喜欢用 . 命令，因为它可以给我即时的视觉反馈。如果需要再次缩进的话，只需再按一次 . 键即可；或者如果按的次数太多，导致缩进过深，按 u 键就可以撤销多余的缩进。在技巧11中，已经用大量篇幅讨论过次数风格与重复风格之间的差异了。

在使用 . 命令重复一条可视模式命令时，它操作的文本数量和上次被高亮选中的文本数量相同。对于面向行的高亮选区来说，这种做法往往符合我们的需要。但对于面向字符的高亮选区来说，这却会产生令人意外的结果。接下来将通过一个例子来说明这一点。

技巧23 只要可能，最好用操作符命令，而不是可视命令

可视模式可能比Vim的普通模式操作起来更自然一些，但是它有一个缺点：在这个模式下，. 命令有时会有一些异常的表现。可以用普通模式下的操作符命令来规避此缺点。

假设想把下面列表中的链接文字转换为大写格式。

visual_mode/list-of-links.html

```
<a
  href="#">one</a
>
<a
  href="#">two</a
>
<a
  href="#">three</a
>
```

可以用 vit 来选择标签里的内容。vit 可被解读为高亮选中标签内部的内容（visually select inside the tag），其中，it 命令是一种被称为文本对象（text object）的特殊动作命令。我们将在技巧52中对其进行详细讲解。

使用可视模式下的命令

在可视模式中，可以选定一个选区然后对其进行操作。本例中，可以使用 U 命令来把所选中的字符转换为大写（参见 :h v_U ⓘ），具体操作参见表4-1。

在转换完第一行后，现在对接下来的两行进行同样的修改。用点范式试一下吧，怎么样？

表4-1 用可视模式下的命令进行大写转换

| 按键操作 | 缓冲区内容 |
|---------|---|
| {start} | < a href="#">one two three |
| vit | one two three |
| U | O NE two three |

执行 j . 命令，把光标移到下一行并重复上次的修改。此命令在第二行工作得很好，但如果再执行一次，最终就会得到这个看起来有点古怪的结果。

```
<a
  href="#">ONE</a
>
<a
  href="#">TWO</a
>
<a
  href="#">THRee</a
>
```

你看到发生什么了吗？当一条可视模式命令重复执行时，它会影响相同数量的文本（参见 `:h visual-repeat` ⓘ）。在本例中，最初的命令影响了一个由3个字母组成的单词。在第二行它依旧工作得很好，因为该行恰好也包含一个由3个字母组成的单词。但是，当我们想对一个由5个字母组成的单词重复此命令时，它只成功转换了其中的前3个字母，留下2个字母未被转换。

使用普通模式下的操作符命令

可视模式下的 `U` 命令有一个等效的普通模式命令：`gU{motion}`（参见 `:h gU` ⓘ）。如果用此命令做第一处修改，就可以用点范式完成后续的修改，如表4-2所示。

表4-2 用普通模式下的操作符命令进行大写转换

| 按键操作 | 缓冲区内容 |
|-------------|---|
| {start} | < a href="#">one two three |
| gUit | O NE two three |
| j. | ONE T WO three |
| j. | ONE T W O T H REE |

结论

这两种方式都只需要4次按键操作：`vitU` 及 `gUit`，但其背后的含义却大相径庭。在可视模式采用的方式中，这4次按键可以被当作两个独立的命令。`vit` 用来选中选区，而 `U` 用来对选区进行转换。与之相反的是，`gUit` 命令可以被当成一个单独的命令，它由一个操作符（`gU`）和一个动作命令（`it`）组成。

如果想使点命令能够重复某些有用的工作，那么最好要远离可视模式。作为一般的原则，在做一系列可重复的修改时，最好首选操作符命令，而不是其对应的可视模式命令。

这并不是说可视模式出局了，它仍然占有一席之地。因为并非每个编辑任务都需要重复执行，对一次性的修改任务来说，可视模式完全够用，并且尽管Vim的动作命令允许进行精确的移动，但有时要修改的文本范围的结构很难用动作命令表达出来，而处理这种情形恰恰是可视模式擅长的。

技巧24 用面向列块的可视模式编辑表格数据

在任何编辑器中，我们都能够操作以行为单位的文本，但以列为单位进行文本操作就需要更为专业的工具了。Vim面向列块的可视模式就提供了这种能力，可以用它来对纯文本表格进行转换。

假设有如下一个纯文本表格。

visual_mode/chapter-table.txt

| Chapter | Page |
|-------------|------|
| Normal mode | 15 |
| Insert mode | 31 |
| Visual mode | 44 |

我们想用管道符画一条竖线来隔开这两列文本，使之看起来更像一个表格。但是在此之前，要先减少两列之间的间隔，使它们不要分得这么开。用面向列块的可视模式可以完成这两处修改，具体做法请参见表4-3。

表4-3 在列间增加分隔竖线

| 按键操作 | 缓冲区内容 |
|---------|--|
| {start} | Chapter █ Page Normal mode 15 Insert mode 31 Visual mode 44 |
| <C-v>3j | Chapter █ Page Normal mode █ 15 Insert mode █ 31 Visual mode █ 44 |
| x... | Chapter █ Page Normal mode 15 Insert mode 31 Visual mode 44 |
| gv | Chapter █ Page Normal mode █ 15 Insert mode █ 31 Visual mode █ 44 |
| r | Chapter Page Normal mode 15 Insert mode 31 Visual mode 44 |

| 按键操作 | 缓冲区内容 |
|------------|--|
| yyp | <pre> Chapter Page C hapter Page Normal mode 15 Insert mode 31 Visual mode 44 </pre> |
| Vr- | <pre> Chapter Page ----- Normal mode 15 Insert mode 31 Visual mode 44 </pre> |

使用 <C-v> 进入列块可视模式，然后向下移动几行光标，选中一列文本。接下来，按 x 键删除此列，并用 . 命令重复删除相同范围的文本。此操作多重复几次直到距右边差不多有两列的距离。

也可以不用 . 命令，而是把光标向右移动两三次，把列选区扩展为块选区，而后只需删除一次即可。不过，我更喜欢在删除时看到即时的视觉反馈，然后再多次重复此操作。

现在，我们已经把所需的两列文本排列到了合适的位置，接下来就可以在这两列文本间画一条竖线了。先用 gv 命令重选上次的高亮选区，然后输入 r|，用管道符替换此选区内的字符。

到了这一步，我们或许也想画一条横线来分隔表头及其下的内容。先快速复制顶行并粘贴一份副本（yyp），然后用连字符替换该行内的所有字符（Vr-）。

技巧25 修改列文本

用列块可视模式可以同时往若干行中插入文本。列块可视模式不仅仅对表格数据有用，在编程时我们也时常受惠于此功能。

例如，对于以下CSS片段。

visual_mode/sprite.css

```
li
.one    a
{ background-image: url
('/images
/sprite.png
'); }
li
.two    a
{ background-image: url
('/images
/sprite.png
'); }
li
.three  a
{ background-image: url
('/images
/sprite.png
'); }
```

假设已经把文件 `sprite.png` 从 `images/` 目录移到了 `components/` 目录，就需要修改每一行的内容，使其指向该文件的新位置。可以使用列块可视模式完成此工作，如表4-4所示。

表4-4 向多行插入文本

| 按键操作 | 缓冲区内容 |
|--------------------|--|
| {start} | li.one a { background-image: url('/i mages/sprite.png'); } li.two a { background-image: url('/images/sprite.png'); } |
| <i>Normal mode</i> | li.three a { background-image: url('/images/sprite.png'); } |
| <C-v>jje | li.one a { background-image: url('/ images /sprite.png'); } li.two a { background-image: url('/ images /sprite.png'); } |
| <i>Visual mode</i> | li.three a { background-image: url('/ images /sprite.png'); } |
| c | li.one a { background-image: url('// sprite.png'); } li.two a { background-image: url('//sprite.png'); } |
| <i>Insert mode</i> | li.three a { background-image: url('//sprite.png'); } |
| components | li.one a { background-image: url('/components/ sprite.png'); } li.two a { background-image: url('//sprite.png'); } |
| <i>Insert mode</i> | li.three a { background-image: url('//sprite.png'); } |
| <Esc> | li.one a { background-image: url('/components /sprite.png'); } li.two a { background-image: url('/components/sprite.png'); } |
| <i>Normal mode</i> | li.three a { background-image: url('/components/sprite.png'); } |

整个过程看起来非常熟悉。先指定想要操作的选区，本例中的高亮选区恰好为方形。按 `c` 键时，所有被选中的文本都消失了，同时进入插入模式。

在插入模式中输入单词“**components**”时，此单词只出现在顶行，下面的两行没什么变化。只有在按 `<Esc>` 返回到普通模式后，才看到刚才输入的文本出现在下面这两行里。

在Vim列块可视模式中，修改命令的表现或许有点怪，它看上去有点不一致。删除操作会同时影响所有被选中的行，但插入操作只影响顶行（至少在处于插入模式的期间）。其他文本编辑器也提供了类似的功能，但是它们会同时更新所有被选中的行，如果你已经习惯了这样的表现（就像我以前一样），那么你会发现Vim的实现不太完美。

不过在实践中，最终的结果没什么区别。因为处于插入模式的时间很短，所以没必要太过惊讶。

技巧26 在长短不一的高亮块后添加文本

列块可视模式在操作由行列组成的方形代码块时表现得很好，然而，它并不仅限于操作方形的文本区域。

我们已经见过以下的JavaScript代码片段。

the_vim_way/2_foo_bar.js

```
var
  foo = 1
var
  bar = 'a'
```

```
'  
var  
  
    foobar = foo + bar
```

这段代码有连续3行，每行的长度各不相同，而我们想在每行结尾添加一个分号。在技巧2中，使用 . 命令解决了此问题，不过，用列块可视模式也可以完成该任务，具体操作参见表4-5。

在进入列块可视模式后，按 \$ 键把选区扩大到每行的行尾。乍一看，人们也许会觉得这很难，因为每一行的长度都是不同的。然而在这个场景中，Vim知道我们是想把选区扩大到选中的这些行的结尾，它会让我们打破方形的限制，创建出一个右边界长短不一的文本选区。

确定好选区后，用 A 命令就可以在每行的结尾添加内容（参见Vim对“i”及“a”键的约定）。此命令让我们进入插入模式，且使光标停留在顶行。处于插入模式期间，任何输入的内容只出现在顶行，然而一旦返回到普通模式，这些修改就会被扩散到其余选中的行上。

表4-5 在列块可视模式中为多行添加分号

| 按键操作 | 缓冲区内容 |
|---------------------|-------------------------------------|
| {start} | var foo = 1 var bar = 'a' |
| <i>Normal mode</i> | var foobar = foo + bar |
| <C-v>jj\$ | var foo = 1 var bar = 'a' |
| <i>Visual-Block</i> | var foobar = foo + bar |

| 按键操作 | 缓冲区内容 |
|--------------------|--|
| A ; | var foo = 1; var bar = 'a' |
| <i>Insert mode</i> | var foobar = foo + bar |
| <Esc> | var foo = 1 ; var bar = 'a'; |
| <i>Normal mode</i> | var foobar = foo + bar; |

Vim 对“i”及“a”键的约定

Vim对于从普通模式切换到插入模式的命令有几个约定，**i** 命令和 **a** 命令都完成此切换，并分别把光标置于当前字符之前或之后，**I** 命令和 **A** 命令的表现类似，只是它们分别把光标置于当前行的开头和结尾。

Vim 对于从列块可视模式切换到插入模式的命令也遵从类似的约定。**I** 命令和 **A** 命令都完成此切换，并分别把光标置于选区的开头和结尾。那 **i** 和 **a** 命令呢，它们在可视模式里干什么？

在可视模式及操作符待决模式中，**i** 和 **a** 键沿用不同的约定。它们会被当作一个文本对象的组成部分，我们将在技巧52中深入探讨文本对象。如果你在列块可视模式里选中了一块区域，并且很奇怪为什么按 **i** 键没进入插入模式，那么换用 **I** 键试一下。

[1] <http://vimcasts.org/e/2>

第5章 命令行模式

初时，先有 ed，ed 为 ex

之父，ex 为 vi 之父，而

vi 为 Vim 之父。

► The Old Testament of Unix

Vim的先祖是vi，正是vi开创了区分模式编辑的范例。相应的，vi奉一个名为ex的行编辑器为先祖，这就是为什么会有 Ex 命令。这些早期UNIX文本编辑器的血脉依旧流淌在现代Vim中，对某些基于行的编辑任务来说，Ex 命令仍然是最佳工具。在本章中，我们将学习如何使用命令行模式，这将为我们的揭示 ex 编辑器的余风遗韵。

技巧27 认识Vim的命令行模式

命令行模式会提示我们输入一条 Ex 命令、一个查找模式，或一个表达式。在本节，我们将结识一些操作缓冲区中的文本的 Ex命令，并学习一些可在此模式中使用的特殊按键映射项。

在按下 : 键时，Vim会切换到命令行模式。这个模式和shell下的命令行有些类似，可以输入一条命令，然后按 <CR> 执行它。在任意时刻，都可以按 <Esc> 键从命令行模式切换回普通模式。

出于历史原因，在命令行模式中执行的命令又被称为 Ex 命令，参见**Vim（及其家族）的词源**。在按 / 调出查找提示符或用 <C-r>= 访问表达式寄存器（参见技巧16）时，命令行模式也会被激活。本节介绍的一些技巧在这些不同的提示符下都适用，不过本节内容主要侧重于Ex 命令。

可以用Ex命令读写文件（:edit 和 :write）、创建新标签页（:tabnew）、分割窗口（:split）、操作参数列表（:prev``/``:next）及缓冲区列表（:bprev``/``:bnext）。事实上，Vim为几乎所有功能都提供了相应的Ex命令（参见 :hex-cmd-index ⓘ 可获得完整列表）。

本节主要关注那些用来编辑文本的 Ex 命令，5-1列出了其中最有用的一些命令。

在这些命令中，绝大部分都可指定操作的范围，将在技巧28中了解这意味着什么。:copy 命令对快速复制一行非常好用，这将在用‘: t’命令复制行 中介绍。:normal 命令提供了一种便捷的方式来自对指定范围内的行做相同的修改，这将在技巧30中介绍。

我们将在第10章学到更多关于 :delete、:yank 及 :put 命令的知识。:substitute 命令和 :global 命令非常强大，所以每个命令都用单独的一章来介绍，详细内容请看第14章和第15章。

表5-1 操作缓冲区文本的 Ex 命令

| 命令 | 用途 |
|--------------------------------------|-------------------|
| : [range] delete [x] | 删除指定范围内的行[到寄存器x中] |
| : [range] yank [x] | 复制指定范围的行[到寄存器x中] |
| : [line] put [x] | 在指定行后粘贴寄存器x中的内容 |

| 命令 | 用途 |
|---|---------------------------------------|
| <code>:[range]copy {address}</code> | 把指定范围内的行拷贝到 {address} 指定的行之下 |
| <code>:[range]move {address}</code> | 把指定范围内的行移动到 {address} 指定的行之下 |
| <code>:[range]join</code> | 连接指定范围内的行 |
| <code>:[range]normal {commands}</code> | 对指定范围内的每一行执行普通模式命令 {commands} |
| <code>:[range]substitute/{pattern}/ {string}/[flags]</code> | 把指定范围内出现 {pattern} 的地方替换为 {string} |
| <code>:</code> <code>[range]global/{pattern}/[cmd]</code> | 对指定范围内匹配 {pattern} 的所有行执行 Ex 命令 {cmd} |

Vim 命令行模式中的特殊按键

在命令行模式中，键盘上的大部分按键都只是简单输入一个字符，这点与插入模式类似。只不过在插入模式中，文本被输入缓冲区里，而在命令行模式中，文本出现在命令行上。另外，在这两种模式中都可以用组合键触发命令。

有些命令在插入模式和命令行模式中可以通用。例如，可以用 `<C-w>` 和 `<C-u>` 分别删除至上个单词的开头及行首，也可以用 `<C-v>` 或 `<C-k>` 来插入键盘上找不到的字符，还可以用 `<C-r>` {register} 命令把任意寄存器的内容插入命令行，就像在技巧15中见过的那样。然而，有些命令行模式中的组合键在插入模式中不存在，我们将在技巧33中结识几个这样的命令。

在命令行提示符下，可以使用的动作命令数量很有限。`<left>` 和 `<right>` 光标键可以一次把光标向左或右移动一个字符，与我们

已经习以为常的普通模式下的大量动作命令相比，这让人感觉极度受限。然而，正如我们即将在技巧34中看到的那样，Vim的命令行窗口提供了构造复杂命令所需的完整编辑能力。

Ex 命令影响范围广且距离远

有时使用 Ex 命令，能比用普通模式命令更快地完成同样的工作。举个例子，普通模式命令一般操作当前字符或当前行，而 Ex 命令却可以在任意位置执行，这意味着无需移动光标就可以使用 Ex 命令做出修改。但使 Ex 命令脱颖而出的最让人赞叹的功能，是它们拥有能够在多行上同时执行的能力。

一般地说，Ex命令操作范围更大，并且能够在一次执行中修改多行。或者可以概括为，Ex命令影响的范围较广并且距离较远。

Vim（及其家族）的词源

ed 是最初的 UNIX 文本编辑器，它编写于图形显示器很稀有的年代，那时源代码通常是打印在纸带上，并在电传终端机^[1]上编辑。在终端上输入的命令被送到大型机上进行处理，每条命令的输出会被打印出来。在那个年代，从终端到大型机之间的连接很慢，以至于一个快速打字员比网络还快，他们输入命令的速度要比命令被发出去处理更快。在这种情况下，ed 能够提供一个简洁的语法变得异常重要。p 被用来打印当前行，而 %p 被用来打印整个文件，皆缘于此。

ed 历经了几代的改进，包括em（意为“editor for mortals”，即“人类的编辑器”）、en，最终到ex^[2]。此时图形显示器已经比较普及了，ex 增加了一个把终端屏幕设置成交互窗口的功能，并在窗口内显示文件的内容。这样，在做修改时实时看到变化成为了可能。此屏幕编辑模式由 :visual 命令激活，其简写为 :vi，这即是 vi 这个名字的由来。

Vim 代表改进版的 vi（vi improved），然而这只是一种谦虚的说法，我实在无法忍受使用标准的 vi。通过查阅 :h vi-

differences ❶，我们可以看到Vim支持而vi不支持的功能列表。Vim对功能的增强是必要的，但另一方面它却仍继承了大量的遗产。这些指导Vim先祖们设计的约束，提供了一个非常高效的命令集，这在今天依然很有价值。

技巧28 在一行或多个连续行上执行命令

很多 Ex 命令可以用 [range] 指定要操作的范围。可以用行号、位置标记或是查找模式来指定范围的开始位置及结束位置。

Ex 命令的优点之一是它可以在某一范围内的所有行上执行。下面这个简短的HTML文本作为示例。

cmdline_mode/practical-vim.html

```
Line 1
<!DOCTYPE html>
  2 <html>

  3   <head><title>
Practical Vim</title></head>

  4   <body><h1>
Practical Vim</h1></body>

  5 </html>
```

我们将使用 `:print` 命令作为演示。这条命令只是简单地在 Vim 命令行下方回显指定行的内容，它不产生什么实际影响，不过可以用它来说明一个范围由哪些行构成。当然，可以试着把以下示例中的 `:print` 换成诸如 `:delete`、`:join`、`:substitute` 或 `:normal` 这样的命令，这样就能真切地感受到 Ex 命令是多么有用。

用行号作为地址

如果输入一条只包含数字的 Ex 命令，那么 Vim 会把这个数字解析成一个地址，并把光标移动到该数字指定的行上。例如，运行下面的命令将跳到文件的首行。

```
⇒ :1
⇒ :print
《
1 <!DOCTYPE html>
```

此文件只包含 5 行内容，如果要跳到文件的末尾，既可以输入 `:5`，也可以用特殊符号 `$`

```
⇒ :$
⇒ :p
《
5 </html>
```

我们在这里使用的是 `:p`，它是 `:print` 命令的简写。实际上，用不着分开执行这两条命令，可以像下面这样把这两条命令合成一条。

```
⇒ :3p
《
3 <head><title>Practical Vim</title></head>
```

此命令会把光标移到第3行，然后显示该行的内容。记住，这里用 `:p`` 命令的目的只是进行讲解。如果执行的是 `:3d`` 命令，只需一条命令就可以跳到第3行并删除此行；而与之等效的普通模式命令，则要先执行 `3G`，再跟着执行 `dd`。因此，从这个例子可以看出，`Ex` 命令执行得要比普通模式命令更快。

用地址指定一个范围

迄今为止，地址只是被当成一个单独的行号，不过也可以用它来指定一个范围，如下例所示。

```
⇒ :2,5p
《

2 <html>
3   <head><title>Practical Vim</title></head>
4   <body><h1>Practical Vim</h1></body>
5 </html>
```

此例会打印第2行到第5行之间的每一行的内容（含第2行及第5行）。注意，运行完这条命令后，光标将停留在第5行。通常，一个范围具有如下的形式。

```
:{start},{end}
```

需注意的是 `{start}` 和 `{end}` 都是地址。到目前为止，我们已经看到过用行号作为地址，然而很快就会看到也能用查找模式或是位置标记作为地址。

符号 `.` 代表当前行的地址。因此，可以很容易地写出一个范围，用以代表从当前位置到文件末尾间的所有行。

```
⇒ :2
⇒ :.,$p
《

2 <html>
3   <head><title>Practical Vim</title></head>
```

```
4 <body><h1>Practical Vim</h1></body>
5 </html>
```

符号 % 也有特殊含义，它代表当前文件中的所有行。

```
⇒ :%p
《

1 <!DOCTYPE html>
2 <html>
3 <head><title>Practical Vim</title></head>
4 <body><h1>Practical Vim</h1></body>
5 </html>
```

这和运行 :1, \$p 是等效的。这种简写形式在和 :substitute 命令一起使用时非常普遍。

```
⇒ :%s/Practical/Pragmatic/
```

上述命令让 Vim 把每行内的第一个“Practical”替换为“Pragmatic”，我们将在第14章学习关于此命令的更多内容。

用高亮选区指定范围

也可以用高亮选区选定一个范围，而不是用数字指定。如果先执行 2G，再跟着执行 VG，就会选中如下一个高亮选区。

```
<!DOCTYPE html>
<html>
  <head><title>Practical Vim</title></head>
  <body><h1>Practical Vim</h1></body>
</html>
```

如果现在按下 : 键，命令行上会预先填充一个范围 : '<, '>。这个范围看起来有点晦涩难懂，不过可以简单地把它理解为一个代表高亮选区的范围。接下来可以输入一条 Ex 命令，使它在每个被选中的行上执行。

```
⇒ : '<, '>p
《

2 <html>
3   <head><title>Practical Vim</title></head>
4   <body><h1>Practical Vim</h1></body>
5 </html>
```

如果只是想对文件的部分内容执行 `:substitute` 命令，用这种方式定义范围会很方便。

符号 `'<` 是代表高亮选区首行的位置标记，`'>` 则代表高亮选区的最后一行（更多关于位置标记的内容，请参见技巧54），这些位置标记即使在退出可视模式后仍然存在。如果尝试在普通模式下直接运行 `: '<, '>p`，它会始终回显上一次高亮选区选中的内容。

用模式指定范围

Vim 也接受以模式作为一条 Ex 命令的地址，如下所示。

```
⇒ : /<html>/, /<\>/html>/p
《

2 <html>
3   <head><title>Practical Vim</title></head>
4   <body><h1>Practical Vim</h1></body>
5 </html>
```

这个范围看起来比较复杂，但实际上它符合范围的一般形式：`{start},{end}`。在本例中，`{start}` 地址是模式 `/<html>/`，而 `{end}` 地址是 `/<\>/html>/`。换句话说，这个范围由 `<html>` 开标签所在的行开始，到对应闭标签所在的行结束。

在此例中，用地址 `:2,5` 也可以获得同样的结果，并且这种表示方式更简洁，不过它也更不可靠。用模式指定范围的话，命令总是对整个 `<html></html>` 范围进行操作，无论这个范围包含多少行都没问题。

用偏移对地址进行修正

假设想对位于 `<html></html>` 之间的每一行都运行一条 Ex 命令，但是不想包括 `<html>` 及 `</html>` 标签所在的行，那么可以为之加上偏移。

```
⇒ :/<html>/+1,/<\html>/-1p
《

3 <head><title>Practical Vim</title></head>
4 <body><h1>Practical Vim</h1></body>
```

偏移的一般形式如下。

```
:{address}+n
```

如果 `n` 被省略，那么缺省偏移量为1。`{address}` 可以是一个行号、一个位置标记，或是一个查找模式。

假设想对由当前行开始的特定几行执行一条命令，那么可以使用相对于当前行的偏移。

```
⇒ :2
⇒ :.,.+3p
```

符号 `.` 代表当前行，所以上例中的 `:.,.+3` 相当于 `:2,5`。

结论

定义范围的语法非常灵活，既可以混合搭配行号、位置标记以及查找模式，也可以对它们加以偏移。下表对用来构建 Ex 命令的地址及范围的符号进行了总结。

| 符号 | 地址 |
|----|----|
|----|----|

| 符号 | 地址 |
|----|-------------------|
| 1 | 文件的第一行 |
| \$ | 文件的最后一行 |
| 0 | 虚拟行，位于文件第一行上方 |
| . | 光标所在行 |
| 'm | 包含位置标记m的行 |
| '< | 高亮选区的起始行 |
| '> | 高亮选区的结束行 |
| % | 整个文件（:1,\$ 的简写形式） |

第0行在文件中并不真实存在，但它作为一个地址，在某些特定场景下会很有用处。特别是在把指定范围内的行复制或移动到文件开头时，可以用它做 `:copy {address}` 及 `:move {address}` 命令的最后一个参数。将在接下来的两个技巧中看到这两条命令的应用实例。

在定义一个 `[range]` 时，它总是代表一系列连续行，不过 `:global` 命令也可以在一系列非连续行上执行Ex命令，我们将在第15章学习这方面的更多知识。

技巧29 使用‘:t’和‘:m’命令复制和移动行

:copy 命令（及其简写形式 **:t**）让我们可以把一行或多行从文档的一部分复制到另一部分，**:move** 命令则可以让把一行或多行移到文档的其他地方。

使用如下购物清单作为演示。

cmdline_mode/shopping-list.todo

```
Line 1
Shopping list
  2
    Hardware Store
      3
        Buy new hammer
          4
    Beauty Parlor
      5
        Buy nail polish remover
          6
        Buy nails
```

用 ‘:t’ 命令复制行

这个购物清单还没完成，我们也要在五金商店（**hardware store**）买些钉子（**nails**）。为完成这个清单，将重用文件的最后一行，即在“**Hardware Store**”下面为之创建一份副本。可以用 **Ex** 命令 **:copy** 轻松地完成这项工作。

| 按键操作 | 缓冲区内容 |
|------|-------|
|------|-------|

| 按键操作 | 缓冲区内容 |
|---------|---|
| {start} | Shopping list H ardware Store Buy new hammer Beauty Parlor Buy nail polish remover Buy nails |
| :6copy. | Shopping list Hardware Store B uy nails Buy new hammer Beauty Parlor Buy nail polish remover Buy nails |

copy命令的格式如下（参见 :h :copy ❶）。

```
: [range] copy {address}
```

在此例中，[range] 是第6行，而 {address} 用的是符号 .，它代表当前行。因此，可以把 :6copy. 命令解读为“为第6行创建一份副本，并放到当前行下方”。

:copy 命令可以简写为两个字母 :co，也可以用更加简练的 ``:t 命令，它是 :copy 命令的同义词。

为了更好地记忆，可以把该命令想成“复制到（copy TO）”。下表展示了 :t 命令的一些应用实例。

| 命令 | 用途 |
|------|--------------|
| :6t. | 把第6行复制到当前行下方 |

| 命令 | 用途 |
|---------|-----------------------------------|
| :t6 | 把当前行复制到第6行下方 |
| :t. | 为当前行创建一个副本（类似于普通模式下的 yyp ） |
| :t\$ | 把当前行复制到文本结尾 |
| :'<,>t0 | 把高亮选中的行复制到文件开头 |

`:t.` 命令会创建一个当前行副本，另外一种做法则是用普通模式的复制和粘贴命令（`yyp`）来达到同样的效果。这两种复制当前行的技术有个需要关注的差别：`yyp` 会使用寄存器，`:t.` 则不会。因此，当我不想覆盖默认寄存器中的当前内容时，有时会使用 `:t.` 来复制行。

在上表中，也可以将 `yyp` 变化一下来复制想要的行，但不管怎样，这都需要一些额外的移动动作。先跳到想复制的行上（`6G`），复制该行（`yy`），快速跳回原先的位置（`<C-o>`），然后再用粘贴命令（`p`）创建一个副本。由此可见，在复制距离较远的行时，`:t` 命令通常更加高效。

在 **Ex 命令影响范围广且距离远** 中，我们已经了解了这个一般规律，即普通模式命令适合在本地进行操作，Ex 命令则可以远距离操作。本节以实例印证了这一规律。

用 ‘:m’ 命令移动行

`:move` 命令看上去和 `:copy` 命令很相似（参见 `:h :move` ⓘ）。

```
: [range]move {address}
```

可以把它简写为一个字母 :m 。假设想把Hardware Store一节移到Beauty Parlor一节的下方，用 :move 就可以实现这一点，如表5-2所示。

在选中高亮选区后，只需简单地执行命令 : '<,'>m\$ 即可。另外还有种做法，可以执行 dGp ，此命令可以分解为：d 删除高亮选区，G 跳转到文件结尾，p 则粘贴刚刚删除的文本。

表5-2 用‘:m’命令对一组进行移动

| 按键操作 | 缓冲区内容 |
|------------|--|
| {start} | Shopping list H ardware Store Buy nails Buy new hammer Beauty Parlor Buy nail polish remover Buy nails |
| Vjj | Shopping list Hardware Store Buy nails ■ Buy new hammer Beauty Parlor Buy nail polish remover Buy nails |
| : '<,'>m\$ | Shopping list Beauty Parlor Buy nail polish remover Buy nails Hardware Store Buy nails B uy new hammer |

记住， '<,'> 代表了高亮选区。因此可以很容易地选中另外一个高亮选区，然后重复执行 : '<,'>m\$ 命令把选中的文本移到文件结

尾。重复上次的 Ex 命令非常简单，只需按 @: 即可（技巧31给出了另一个例子），所以这里采取的方式与使用普通模式命令相比，在重复执行时会更方便。

技巧30 在指定范围上执行普通模式命令

如果想在一系列连续行上执行一条普通模式命令，可以用 :normal 命令。此命令在与 . 命令或宏结合使用时，只需花费很少的努力就能完成大量重复性任务。

想一下在技巧2中遇到过的例子，我们想在一系列行后添加一个分号。使用点范式让我们迅速完成了这项工作。但是在那个例子里，只需对连续的3行做此修改。如果不得不做50次同样的修改会怎么样呢？如果还用点范式的话，得按50次 j.，总共得100次按键动作！

这里有一种更好的方法。将在下面文件的每行后都添加一个分号，以此作为演示。为节省空间，此处只列出了5行内容，然而你可以想象这里有50行，那么这种方法看起来就颇具诱惑了。

cmdline_mode/foobar.js

```
var
  foo = 1
var
  bar = 'a
'
var
  baz = 'z
'
var
```

```
    foobar = foo + bar
var

    foobarbaz = foo + bar + baz
```

我们像之前做的那样，首先修改第一行。

| 按键操作 | 缓冲区内容 |
|----------|--|
| {start} | <code>v ar foo = 1</code> <code>var bar = 'a'</code> <code>var baz = 'z'</code> <code>var foobar = foo + bar</code> <code>var foobarbaz = foo + bar + baz</code> |
| A ;<Esc> | <code>var foo = 1;</code> <code>var bar = 'a'</code> <code>var baz = 'z'</code> <code>var foobar = foo + bar</code> <code>var foobarbaz = foo + bar + baz</code> |

接下来，不用一行一行地执行 `. 命令`，而是使用 `Ex 命令` `:normal` 对整个范围内的所有行同时执行 `. 命令`。

| 按键操作 | 缓冲区内容 |
|------|--|
| jVG | <code>var foo = 1;</code> <code>var bar = 'a'</code> <code>var baz = 'z'</code> <code>var foobar = foo + bar</code> <code>var foobarbaz = foo + bar + baz</code> |

| 按键操作 | 缓冲区内容 |
|-----------------------------------|--|
| <code>:'<,'>normal .</code> | <pre>var foo = 1; var bar = 'a'; var baz = 'z'; var foobar = foo + bar; var foobarbaz = foo + bar + baz;</pre> |

`:'<,'>normal .` 命令可以解读为“对高亮选区中的每一行，对其执行普通模式下的 `. 命令`”。无论是操作5行还是50行文本，这种方法都能出色地完成任务，更棒的是我们甚至都不需要计算行数，在可视模式中选中这些行使我们摆脱了计数的负担。

这个例子使用 `:normal` 执行 `. 命令`，但是也可以用这种方式执行任意其他的普通模式命令。例如，可以用如下命令解决上面的问题。

```
⇒ :%normal A;
```

符号 `%` 代表整个文件范围，因此 `:%normal A;` 告诉 Vim 在文件每行的结尾都添加一个分号。在做此修改时会切换到插入模式，但是在修改完后，Vim会自动返回到普通模式。

在执行指定的普通模式命令之前，Vim会先把光标移到该行的起始处。因此在执行时，用不着担心光标的位置。例如，下面这条命令可以把整个JavaScript文件注释掉。

```
⇒ :%normal i//
```

虽然用 `:normal` 命令可以执行任意的普通模式命令，但是我发现当它和 Vim 的重复命令结合在一起时，最为强大，既可以用 `:normal .` 应对简单的重复性工作，也可以用 `:normal @q` 应对较复杂的任务。具体的实例参见技巧68和技巧70。

在 **Ex 命令影响范围广且距离远** 中，我们说过 Ex 命令可以一次修改若干行。`:normal` 命令则让我们可以把具有强大表现力的 Vim 普通模式命令与具有大范围影响力的 Ex 命令结合在一起，这种结合真的是珠联璧合！

对本节所涉及问题的另外一种解决方案，请参见技巧26。

技巧31 重复上次的 Ex 命令

`.` 命令可以重复上次的普通模式命令。然而，如果想重复上次的 Ex 命令，得使用 `@:` 才行。知道如何回退上次的命令永远是有价值的，因此本节也会讨论这一点。

在第1章中，我们见识过如何用 `.` 命令重复上次的修改。但是，`.` 命令不会重复由 Vim 命令行中做出的修改。作为替代，可以用 `@:` 来重复上次的 Ex 命令（参见 `:h ``@:` ❶）。

例如，下面两条命令在遍历缓冲区列表的条目时非常有用，用 `:bn[ext]` 可以在列表中逐项正向移动，而 `:bp[revious]` 命令进行反向移动（技巧37详细讨论了缓冲区列表）。假设缓冲区列表中有大约十几个条目，而我们打算逐个查看每个缓冲区，因此可以输入一次下面的命令。

```
⇒ :bnext
```

然后再用 `@:` 重复执行此命令。留意一下这和运行宏的相似之处（参见**通过执行宏来回放命令序列**），另外也需注意，`:` 寄存器总是保存着最后执行的命令行命令（参见 `:h quote_:` ❶）。在运行过一次 `@:` 后，后面就可以用 `@@` 命令来重复它。

假设我们按得忘乎所以，执行了太多次 `@:` 命令以致于错过了目标。那要怎样才能改变方向往回跳呢？当然，可以执行 `:bprevious`

命令，但是想想如果以后再次执行 @: 命令会发生什么？没错，它会反向遍历缓冲区列表，恰恰与最初的方向相反。这会把人搞糊涂的。

在这种情况下，更好的选择是使用 <C-o> 命令（参见技巧56）。每次运行 :bnext 命令（或用 @: 命令重复执行它）时，它都会在跳转列表中添加一条记录，而 <C-o> 命令会回到跳转列表的上条记录。

可以执行一次 :bnext，然后用 @: 重复任意多次；如果想往回跳，就用 <C-o> 命令。这样一来，如果接下来还想继续正向遍历缓冲区列表，就可以继续用 @: 命令。请牢记技巧4中提到的口诀：执行、重复、回退。

Vim 为几乎所有功能都提供了相应的 Ex 命令。虽然用 @: 总是可以重复上一条 Ex 命令，但如果想回退其影响，却没有这种直截了当的方式。用本节提到的 <C-o> 命令，也能够回退 :next、:``cnext、:tnext 等命令的执行结果；然而对于5-1中列出的 Ex 命令，则要用 u 键才能撤销其影响。

技巧32 自动补全 Ex 命令

如同在 shell 中一样，在命令行上也可以用 <Tab> 键自动补全命令。

Vim 在选取 Tab 补全的补全项时非常智能，它会检查命令行上已经输入的上下文，然后再构建合适的补全列表。例如，可以这样输入：

```
=> :col<C-d>
《
colder colorscheme
```

`<C-d>` 命令会让Vim显示可用的补全列表（参见 `:h c_CTRL-D` ❶）。另外，如果多次按 `<Tab>` 键，命令行上会依次显示 `colder`、`colorscheme`，然后再回到最初的 `col`，如此循环往复。要想反向遍历补全列表，可以按 `<S-Tab>`。

假设想改配色方案，但是不太记得要用的配色方案的名称，可以用 `<C-d>` 命令列出所有的可用选项。

```
⇒ :colorscheme <C-d>  
《
```

```
blackboard  desert      morning      shine  
  blue      elflord      murphy      slate  
darkblue    evening      pablo      solarized  
default     koehler      peachpuff    torte  
delek       mac_classic    ron         zellner
```

这一次，`<C-d>` 基于可用的配色方案显示一个补全列表。如果想激活 `solarized` 方案，只需输入字母“so”，然后按 `Tab` 键即可补全此命令。

在很多场景中，Vim的 `Tab` 补全都能做出正确的选择。如果输入了一个以文件路径作为参数的命令（如 `:edit` 或 `:write`），那么 `<Tab>` 会用当前工作目录中的目录或文件名补全。在 `:tag` 命令中，它会自动补全标签名；而在 `:set` 及 `:help` 命令中，它可以补全 Vim 的每一个设置选项。

甚至在创建自定义 Ex 命令时，也能够定义该命令的 `Tab` 键补全行为。要想了解更多，请查阅 `:h :command-complete` ❶。

在多个补全项间选择

当 Vim 只找到一个 `Tab` 补全项时，它会直接使用整个补全项。但是如果 Vim 找到了多个补全项，那么会有几种做法。缺省情况下，首次按下 `Tab` 键时，Vim 会用第一个补全项补全，以后每按一下 `Tab` 键，就会依次遍历剩余的补全项。

调整 `'wildmode'` 选项可以自定义补全行为（参见 `:h 'wildmode'` ⓘ）。如果习惯用 `bash shell` 的方式工作，那么下面的设置会满足你的需要。

```
set wildmode  
=longest,list
```

如果习惯于 `zsh` 提供的自动补全菜单，或许会想试试这个。

```
set wildmenu  
set wildmode  
=full
```

当 `'wildmenu'` 选项被启用时，Vim 会提供一个补全导航列表。可以按 `<Tab>`、`<C-n>` 或 `<Right>` 正向遍历其列表项，也可以用 `<S-Tab>`、`<C-p>` 或 `<Left>` 对其进行反向遍历。

技巧33 把当前单词插入命令行

即使是在命令行模式下，Vim 也始终知道光标位于何处以及哪个分割窗口处于活动状态。为节省时间，可以把活动窗口中的当前单词（或字串）插入命令行中。

在 Vim 的命令行下，`<C-r><C-w>` 映射项会复制光标下的单词并把它插入命令行中。可以利用这一功能减少击键的次数。

假设有下面这段代码中的变量 `tally` 重命名为 `counter`。

cmdline_mode/loop.js

```
var
  tally;
for
  (tally=1; tally <= 10; tally++) {
    // do something with tally
  };
```

把光标移到单词 `tally` 上后，用 `*` 命令可以查找它出现的每处地方（`*` 命令等效于输入 `/\<<C-r><C-w>\><CR>` 序列，关于 `\<` 和 `\>` 在模式中的作用，请参见技巧77的讨论）。

| 按键操作 | 缓冲区内容 |
|-----------------|---|
| {start} | var t ally; for (tally=1; tally <= 10; tally++) { // do something with tally }; |
| * | var tally; for (tally =1; tally <= 10; tally ++) { // do something with tally }; |
| cw counter<Esc> | var tally ; for (counter =1; tally <= 10; tally ++) { // do something with tally }; |

当按下 `*` 键时，光标会正向跳到下一处匹配项，不过光标始终停留在相同的单词上。接下来，可以输入 `cw counter<Esc>` 对其进行修

改。

然后用 `:substitute` 命令完成其余的修改。由于光标已经在单词“**counter**”上了，因此无需再次输入它，而是直接用 `<C-r><C-w>` 映射项把它插入替换域。

```
⇒ :%s//<C-r><C-w>/g
```

这条命令看起来没省多少事，但是用两次按键就能插入一个单词不算太糟。此处也用不着输入查找模式，而这要感谢 `*` 命令。要知道为什么可以像上面这样将查找域留空，请参考技巧91。

`<C-r><C-w>` 用于插入光标下的单词，而如果想插入光标下的字符串（参见技巧 49 的说明），可以用 `<C-r><C-a>`，更多细节请参见 `:h c_CTRL-R_CTRL-W` ①。虽然本例是以 `:substitute` 命令作为示例的，但实际上这些映射项可用于任意 Ex 命令。

这里介绍另一种应用场景。试着打开你的 `vimrc` 文件，把光标移到其中的一项设置上，然后输入 `:help <C-r><C-w>` ②，就可以查阅该设置的文档了。

技巧34 回溯历史命令

Vim 会记录命令行模式中执行过的命令，并提供了两种方式回溯这些命令，用光标键回滚之前的命令或调出命令行窗口查看先前的命令。

Vim 会记录命令行模式下的命令历史，并且可以很容易地回溯之前的命令，因此对于比较长的 Ex 命令来说，不用在命令行中多次输入它。

先按 `:` 键切换到命令行模式，在保持提示符为空的情况下按 `<Up>` 键，此时最后执行的那条 `Ex` 命令就会被填充到命令行上。紧接着按 `<Up>` 键，就可以回到更早的 `Ex` 历史命令；按 `<Down>` 键，则会沿相反方向滚动。

现在，尝试先输入 `:help` ^❶，然后按 `<Up>` 键遍历之前的 `Ex` 命令。这一次，`Vim` 不会显示所有的历史命令，而是会对列表进行过滤，只有以单词“`help`”开头的 `Ex` 命令才会被包含在列表中。

`Vim` 缺省会记录最后20条命令，对内存越发便宜的现代计算机来说，保存更多历史命令只是小菜一碟，因此可以修改 ```history``` 选项，以提高其保存的上限。可以试着把下面这行内容加入 `vimrc` 文件。

```
set history=200
```

注意：

命令历史不仅是为当前编辑会话记录的，这些历史即使在退出 `Vim` 再重启之后仍然存在（参见 `:h viminfo` ^❶），因此提高历史记录数目非常有价值。

`Vim` 不仅会记录 `Ex` 命令的历史，还会为查找命令单独保存一份历史记录。在按 `/` 调出查找提示符后，用 `<Up>` 和 `<Down>` 键可以正向或反向遍历之前的查找记录。从本质上讲，查找提示符只是命令行模式的另一种形式。

结识命令行窗口

像插入模式一样，命令行模式适合从头开始构建命令，但它却不是编辑文本的好地方。

假设我们正在写一个简单的 `Ruby` 脚本，然后发现每做出一个修改时，都会执行下面两条命令。

```
⇒ :write  
⇒ :!ruby %
```

在接连执行了几次这两条命令后，我们意识到可以简化工作过程，把这两条命令合为一条。这样，以后就可以从历史中选择该完整命令并再次执行。

⇒ :write | !ruby %

这些命令都已经在历史中了，所以不必从头输入整条命令。但要怎样才能把历史中的两条记录合并成一条呢？请输入 q:，先结识一下命令行窗口（参见 :h cmdwin ⓘ）。

命令行窗口就像是一个常规的 Vim 缓冲区，只不过它的每行内容都对应着命令历史中的一个条目。可以用 k 及 j 键在历史中向前或向后移动，也可以用 Vim 的查找功能查找某一行。在按下 <CR> 键时，会把当前行的内容当成 Ex 命令加以执行。

命令行窗口的好处在于它允许使用 Vim 完整的、区分模式的编辑能力来修改历史命令。可以用任何习以为常的动作命令进行移动，也可以在高亮选区上操作，或是切换到插入模式中，甚至还能对命令行窗口中的内容执行 Ex 命令。

在按 q: 调出命令行窗口后，可以像下面这样解决问题。

| 按键操作 | 缓冲区内容 |
|-------------|--------------------|
| {start} | w rite !ruby % |
| A _ <Esc> | write !ruby % |
| J | write █!ruby % |

| 按键操作 | 缓冲区内容 |
|-----------------|---------------------|
| :s/write/update | u pdate █ !ruby % |

修改完后，按 <CR> 就会执行 `:update | !ruby%` 命令，就好像在命令行输入了这条命令一样。

当命令行窗口处于打开状态时，它会始终拥有焦点。这意味着，除非关闭命令行窗口，否则无法切换到其他窗口。要想关闭命令行窗口，可以执行 `:q` 命令（就像关闭普通 Vim 窗口那样），或是按 <CR>。

注意：

在命令行窗口内按 <CR> 时，该命令在活动窗口的上下文中执行。活动窗口是指在调出命令窗口前，处于活动状态的那个窗口。当命令行窗口处于打开状态时，Vim 并不会提示哪个窗口是活动窗口，因此如果使用了分割窗口，就需要特别留意。

假设正在命令行上构建一条 Ex 命令，做到一半时，才意识到需要更强大的编辑能力，这时该怎么办呢？当处于命令行模式下时，可以用 <C-f> 映射项切换到命令行窗口中，此前已经输入命令行上的内容仍然会得以保留。下表总结了打开命令行窗口的几种方式。

| 命令 | 动作 |
|----------|------------------|
| q/ | 打开查找命令历史的命令行窗口 |
| q: | 打开 Ex 命令历史的命令行窗口 |
| <Ctrl-f> | 从命令行模式切换到命令行窗口 |

`q:` 命令和 `:q` 命令很容易混淆。我敢肯定我们都曾经不小心打开过命令行窗口，而实际上我们只是想退出 Vim。这的确让人羞赧，因为这个功能是如此的有用，但是很多人在他们第一次（意外）遭遇它时却感觉很沮丧。要看命令行窗口的另一个应用实例，请跳到技巧 85。

技巧35 运行Shell命令

不用离开 Vim 就能方便地调用外部程序。更棒的是，还可以把缓冲区的内容作为标准输入发送给一个外部命令，或是把外部命令的标准输出导入缓冲区里。

本节讨论的命令在终端 Vim 中工作得最好。如果在运行 GVim（或 MacVim），那么命令运行得也许没那么顺畅。这没什么好奇怪的，如果 Vim 自身在 shell 里运行，那把工作委派给 shell 也会容易得多。GVim 在某些其他方面做得更好一些，但是终端 Vim 在这件事上则更有优势。

执行 Shell 中的程序

在 Vim 的命令行模式中，给命令加一个叹号前缀（参见 `:h :!` ①）就可以调用外部程序。例如，如果想查看当前目录的内容，可以运行下面的命令。

```
=> :!ls

《

duplicate.todo          loop.js
emails.csv              practical-vim.html
foobar.js               shopping-list.todo
history-scrollers.vim

Press ENTER or type command to continue
```

注意区分 `:!ls` 和 `:ls` 的不同之处。前者调用的是 `shell` 中的 `ls` 命令，而 `:ls` 调用的是 Vim 的内置命令，用来显示缓冲区列表的内容。

在 Vim 的命令行中，符号 `%` 代表当前文件名（参见 `:h cmdline-special` ⓘ）。在运行那些操作当前文件的外部命令时，可以使用它。例如，如果正在编辑某个 Ruby 文件，那么可以用下面的方式执行此文件。

```
⇒ :!ruby %
```

Vim 也提供了一组文件名修饰符，让我们可以从当前文件名中提取出诸如文件路径或扩展名之类的信息（参见 `:h filename-modifiers` ⓘ），技巧45中有一个使用这些修饰符的例子。

`:{cmd}` 这种语法适用于执行一次性命令，但是如果想在 `shell` 中执行几条命令要怎么做？对于这种情况，可以执行 Vim 的 `:shell` 命令来启动一个交互的 `shell` 会话（参见 `:h :shell` ⓘ）。

```
⇒ :shell

⇒ $ pwd

《

/Users/drew/books/PracticalVim/code/cmdline_mode

⇒ $ ls

《

duplicate.todo      loop.js
emails.csv          practical-vim.html
foobar.js            shopping-list.todo
history-scrollers.vim

⇒ $ exit
```

用 `exit` 命令可以退出此 `shell` 并返回 Vim。

把Vim置于后台

`:shell` 命令是 Vim 提供的一个功能，它可以切换到一个交互 shell 中。但是，如果 Vim 自身是在终端中运行的，那么也能直接访问终端内置的 shell 命令。例如，`bash shell` 支持作业控制，让我们可以暂停一个作业，把它放到后台，然后在稍后某个时间再把它调回前台继续运行。

假设正在 `bash shell` 中运行 Vim，然后需要执行一些 shell 命令。可以先按 `Ctrl-z` 挂起 Vim 所属的进程，并把控制权交还给 `bash`。此时 Vim 进程在后台处于挂起状态，让我们可以像往常一样与 `bash` 会话进行交互。运行下面这条命令可以查看当前的作业列表。

```
⇒ $ jobs
《
[1]+  Stopped          vim
```

在 `bash` 中，可以用 `fg` 命令唤醒一个被挂起的作业，把它移到前台。这会让 Vim 恢复成挂起前的状态。`Ctrl-z` 和 `fg` 命令比 Vim 提供的 `:shell` 和 `exit` 命令更加方便快捷。要想了解更多信息，请运行 `man bash`，然后阅读作业控制（`job control`）一节。

把缓冲区内容作为标准输入或输出

在用 `:{cmd}``` 时，Vim 会回显 {cmd} 命令的输出。如果命令的输出很少或没有输出，这工作得很好；但如果命令会产生大量输出，这样回显用处不大。另外一种做法是可以用 `:read !{cmd}` 命令，把 {cmd} 命令的输出读入当前缓冲区中（参见 `:h :read!``` ①）。

`:read !{cmd}` 命令让我们把命令的标准输出重定向到缓冲区。正如你所期望的一样，`:write !{cmd}` 做相反的事。它把缓冲区内容作为指定 `{cmd}` 的标准输入（参见 `:h :write_c` ❶），跳到技巧46可以看到此功能的一个应用实例。

根据叹号在命令行上的位置不同，它的含义也不大相同。比较以下命令。

```
⇒ :write !sh
⇒ :write ! sh
⇒ :write! sh
```

前两个命令都会把缓冲区的内容传给外部的 `sh` 命令作为标准输入，而最后一条命令调用 `:write!` 命令把缓冲区内容写到一个名为 `sh` 的文件，这里的叹号会让Vim覆盖任何已存的 `sh` 文件。正如你看到的那样，叹号放的位置不同，命令的作用也大相径庭。因此，在构建这类命令时要多加小心。

`:write !sh` 命令的作用是在shell中执行当前缓冲区中的每行内容，查阅`:h `` ``rename-files``` ❶可看到该命令的一个绝佳示例。

使用外部命令过滤缓冲区内容

当给定一个范围时，`:!{cmd}` 命令就具有了不同的含义。由 `[range]``` 指定的行会传给 `{cmd}` 作为标准输入，然后又会用 `{cmd}` 的输出覆盖 `[range]` 内原本的内容。换一种说法就是 `[range]` 内的文本会被指定的 `{cmd}` 过滤（参见 `:h :range!` ❶）。Vim把过滤器定义为“一个由标准输入读取文本，并对其进行某种形式的修改后输出到标准输出的程序”。

作为演示，将用外部的 `sort` 命令对下列 CSV 文件中的记录进行排序。

cmdline_mode/emails.csv

```
first name,last name,email
john,smith,john@example.com
drew,neil,drew@vimcasts.org
jane,doe,jane@example.com
```

我们想基于第二个字段“姓氏”来重排这些记录。可以用 `-``t'``,``'` 参数告诉 `sort` 命令，这些记录以逗号分隔，然后再用 `-``k2` 参数指定按第二个字段进行排序。

因为文件的第一行是标题信息，我们想把它们保留在文件顶部，因此需要用范围 `:2,$` 把它排除在排序范围之外。下列命令将完成我们想要的功能。

```
⇒ :2,$!sort -t',' -k2
```

现在 CSV 文件中的内容就是按姓氏排序的了。

```
first name,last name,email
jane,doe,jane@example.com
drew,neil,drew@vimcasts.org
john,smith,john@example.com
```

Vim 提供了一种方便的快捷方式来设置 `: [range]! {filter}` 命令中的范围。可以用 `! {motion}` 操作符切换到命令行模式，并把指定 `{motion}` 涵盖的范围预置在命令行上（参见 `:h !` ❶）。例如，如果把光标移到第2行，然后执行 `!G`，**Vim** 就会打开命令行并把范围 `:.,$` 预置在命令行上。虽然此后仍需输入剩下的 `{filter}` 命令，但这毕竟节省了部分工作。

结论

在 **Vim** 中操作时，可以很方便地调用 `shell` 命令。下表选取了最有用的一些调用外部命令的方式。

| 命令 | 用途 |
|----------------------|---------------------------------------|
| :shell | 启动一个shell (输入exit返回 Vim) |
| !!{cmd} | 在shell 中执行 {cmd} |
| :read !{cmd} | 在shell 中执行 {cmd} ， 并把其标准输出插入光标下方 |
| :[range]write !{cmd} | 在 shell 中执行 {cmd} ， 以 [range] 作为其标准输入 |
| :[range]!{filter} | 使用外部程序 {filter} 过滤指定的 [range] |

Vim 对某些外部命令会另眼相待。例如，make 及 grep 在 Vim 中都有包装命令，这些命令不仅执行起来更方便，而且Vim会将它们的输出解析、导入 quickfix 列表中。将在第17章和第18章用很大篇幅介绍这两条命令。

技巧36 批处理运行Ex命令

如果要执行一连串Ex命令，可以把它们置于脚本之中，从而节省工作量。当再想执行那一组命令时，只需加载脚本文件即可，而无需逐条输入这些命令。

以下内容来源于Vimcasts.org归档网页中前两部主题的连接。

cmdline_mode/vimcasts/episodes-1.html


```
<li>

    <a
href="/episodes
/show-invisibles
/">
        Show invisibles
    </a>

<li>

<li>

    <a
href="/episodes/tabs-and-spaces
/">
        Tabs and Spaces
    </a>

</li>

</ol>
```

我们想把其内容转成纯文本格式，标题在前，URL在后。

cmdline_mode/vimcasts-episodes-1.txt

```
Show invisibles: http://vimcasts.org/episodes/show-invisibles/
Tabs and Spaces: http://vimcasts.org/episodes/tabs-and-spaces/
```


假设需要对一组格式相似的文件进行这种转换，来看一下几种不同的操作方法。

逐条执行Ex命令

其实用一条 `:substitutue` 命令就可以实现这种格式转换，不过我更倾向于用几条小命令来完成。以下Ex命令序列就是一种可行的方案。

```
⇒ :g/href/j

⇒ :v/href/d

《

8 fewer lines
⇒ :%norm A: http://vimcasts.org

⇒ :%norm yi"$p

⇒ :%s/\v^[^\>]+\>\s//g
```

不理解这些命令也没关系，这不会影响你对本技巧的学习。不过如果你感兴趣的话，下面是对这些命令的简要介绍。`:global` 命令和 `:vglobal` 命令结合在一起使用，用于把此文件缩减成两行，其中包含了我们所需要的内容，只不过前后次序是颠倒的（技巧 99）；而 `:normal` 命令会在行尾加上 Vimcast 网站的根链接（技巧 30）；最后的 `:substitute` 命令会删除 `` 标签。就像我常说的那样，理解命令的最佳途径就是自己实践一下。

把Ex命令存成脚本并加载

除了可以逐条执行命令，还可以把它们存成一个文件，比如存为 **batch.vim**（使用扩展名 **.vim** 可以使 Vim 显示正确的语法高亮）。文件中的每一行都对应前文中的一条 Ex 命令。在这种情况下，不必为每一行加上前缀字符 **:**。在把 Ex 命令保存到文件时，我个人更倾向于使用命令的全名，因为此时更关注脚本的易读性，而不是节省按键次数。

cmdline_mode/batch.vim

```
global/href/join

vglobal/href/delete

%normal A: http://vimcasts.org
%normal yi"$p
%substitute/\v^[^\>]+\>\s//g
```

可以用 **:source** 来执行 **batch.vim** 脚本（参见 **:h source**）。脚本中的每一行都会被当成一条 Ex 命令执行，就像在 Vim 的命令行中输入这些命令一样。在之前的场景中，你也许已经见识过 **:source`** 命令了：它常用于在运行时加载 **vimrc** 文件（更多信息，请参见将配置信息存至 **vimrc** 文件）。

我建议你亲自试一下。这些代码可以从位于 **Pragmatic Bookshelf** 网站的 **Practical Vim** 主页下载。进入 **cmdline_mode** 目录，就可以看到 **batch.vim** 以及 **episodes-1.html**，然后再打开 Vim。

```
⇒ $ pwd

《

~/dnvim2/code/cmdline_mode
⇒ $ ls *.vim

《
```

```
batch.vim          history-scrollers.vim
⇒ $ vim vimcasts/episodes-1.html
```

现在就可以执行此脚本了：

```
⇒ :source batch.vim
```

仅用这一条命令，就可以执行**batch.vim**中的所有Ex命令了。如果你改变了主意，只需按下 **u** 键即可让文档完好如初。

用此脚本修改多个文件

如果脚本只执行一次，那么把 Ex 命令存成文件没有多大意义。只有想多次运行一组 Ex 命令，这一技巧才彰显其价值。

随书提供的代码例库包含了一些格式与**episodes-1.html**相同的文件。请确保在启动Vim之前切换到**cmdline_mode**目录。

```
⇒ $ pwd

《

~/dnvim2/code/cmdline_mode
⇒ $ ls vimcasts

《

episodes-1.html episodes-2.html episodes-3.html
⇒ $ vim vimcasts/*.html
```

使用通配符启动Vim时，匹配该通配符的所有文件会被加入 Vim 的参数列表里。可以一个个地遍历这些文件，逐一执行batch.vim。

```
⇒ :args

《

[vimcasts/episodes-1.html] vimcasts/episodes-2.html
vimcasts/episodes-3.html
⇒ :first

⇒ :source batch.vim

⇒ :next

⇒ :source batch.vim

《

etc.
```

不过更棒的方法是使用 `:argdo` 命令。

```
⇒ :argdo source batch.vim
```

只需这一条命令，就可以对参数列表里的每个文件执行 batch.vim 中的Ex命令了。

我之所以用几种不同的 Ex 命令来展示这一技术，只是为了说明这一技巧适用的可能性。在实践中，如果发现要一遍又一遍地执行某几条 `:substitute`` 命令时，我常常会用脚本完成。在执行完 batch.vim后，我通常会把它删掉；但如果我认为将来可能会再用到的话，也会将其纳入版本控制。

[1] <http://en.wikipedia.org/wiki/Teleprinter>

[2] http://www.theregister.co.uk/2003/09111/biu-joys.greatert_gifo/

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在 里填入可使用的积分数值，即可扣减相应金额。

特别优惠

购买本电子书的读者专享异步社区优惠券。使用方法：注册成为社区用户，在下单购书时输入“57AWG”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



软技能：代码之外的生存指南

[美]约翰 Z. 森梅兹 (John Z. Sonmez) (作者) 王小刚 (译者) 杨海玲 (责任编辑)

分享

6 推荐

想读

9.0K 阅读

这是一本真正从“人”（而非技术也非管理）的角度关注软件开发人员自身发展的书。书中论述的内容既涉及生活习惯，又包括思维方式，凸显技术中“人”的因素，全面讲解软件行业从业人员所需知道的所有“软技能”。

本书聚焦于软件开发人员生活的方方面面，从揭秘面试的流程到精耕细作出一份杀手级简历，从创建大受欢迎的博客到打造你的个人品牌，从提高自己工作效率到与如何与“拖延症”做斗争，甚至包括如何投资不动产，如何关注自己的健康。

本书共分为职业篇、自我营销篇、学习篇、生产力篇、理财篇、健身篇、精神篇等七篇，概括了软件行业从业人员所需的“软技能”。

● 纸质版 ~~¥59.00~~ **¥46.02 (7.8折)**

● 电子版 **¥35.00**

● 电子版 + 纸质版 **¥59.00**

现在购买

下载PDF样章

配套文件下载

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在这里一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的

乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



异步社区



微信订阅号



微信服务号



官方微博



QQ群：368449889

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@人邮异步社区，@人民邮电出版社-信息技术分社

投稿&咨询: contact@epubit.com.cn